Towards a Formal Verification of the Trusted Platform Module

BY

©2013
Brigid R. Halling

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Chairperson Dr. Perry Alexander

Dr. Andy Gill

Dr. Fengjun Li

Date Defended: May 3, 2013

The Thesis Committee for Brigid R. Halling
certifies that this is the approved version of the following thesis:

Towards a Formal Verification of the Trusted Platform Module

Chairperson Dr. Perry Alexander

Date Approved:

Abstract

The Trusted Platform Module (TPM) serves as the root-of-trust in a trusted computing environment, and therefore warrants formal specification and verification. This thesis presents results of an effort to specify and verify an abstract TPM 1.2 model using PVS that is useful for understanding the TPM and verifying protocols that use it. TPM commands are specified as state transformations and sequenced to represent protocols using a state monad. Preconditions, postconditions, and invariants are specified for individual commands and validated. All specifications are written and verified automatically using the PVS decision procedures and rewriting system.

Acknowledgements

I would like to thank my advisor, Dr. Perry Alexander, for his guidance and help in mastering the ways of the TPM, in addition to being a fantastic teacher both in and out of the classroom. Additionally, many thanks to Drew, Evan, Megan, and Nick for teaching me the ropes of grad school.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's society, with the use of computers and smart phones, we are often in contact with people and machines that we have never seen – and likely never will. Partly due to necessity, partly naiveté , we *trust* our own machine and that of our correspondent. So what does it mean for a system to be trusted? We are often aware of the security aspects of our interactions by having strong passwords and avoiding suspicious links. However, trust is different from security. For a system to be trusted, it must be able to strongly identify itself, operate unhindered, and have experience or record of good, consistent behavior [25]. When it comes to trust, we care about the participants in the interaction. A trusted system is one that is predictable [31]. By this definition, however, a trusted system is not necessarily a good system. But, once we have reliable predictions, we can decide which of these systems we want to interact with.

At the heart of trusted computing [7] is the need to appraise a remote system in a trusted fashion. In this process, known as *remote attestation* [10, 11, 18], an external appraiser sends an attestation request to an appraisal target and receives a quote used to assess the remote system's state. To achieve its goal, the appraiser

must not only analyze the contents of the quote, but also assess the trustworthiness of the information it contains.

The Trusted Computing Group (TCG), an industry initiative created with the intent of developing trusted computing, designed the industry specification for a hardware coprocessor known as the Trusted Platform Module [1]. The TPM provides in protected storage of keys and measurements of data that report on the state of the platform sent within quotes for remote attestation.

The TPM was designed to report on the state of its system using three roots of trust: root of trust for reporting, root of trust for storage, and root of trust for measurement. Each TPM has an Endorsement Key (EK) that uniquely identifies that specific TPM. Building off of the assurance that the TPM is the only entity with access to the private EK, the TPM uses that key as the root key of trust for reporting system information. As the root of trust for storage, the TPM creates a Storage Root Key (SRK), that establishes trust in the local platform. Within a TPM, measurement refers to taking a cryptographic hash of the item to be measured. The root of trust for measurement comes from the trusted implementation of this measurement with a hash algorithm [25]. With the use of these roots of trust, the TPM can send remote attestation quotes and serve as a building block in protocols implementing trusted computing.

As with any hardware or software implementation, the implementations of the TPM have been rigorously tested to verify the specification has been correctly implemented. However, with a system as critical as this, testing implementation is not good enough. What we really *need* to know is that our trust in this design is well-founded. That can't be shown by testing. It must be proved.

This thesis shows the work we have done to formally verify the TPM. Specif-

ically, we started with the TCG's documentation that describes the data and functionality necessary to the TPM. We have taken that specification and turned it into an abstract model about which we can prove specific properties, most importantly the correctness of the remote attestation protocol. We use PVS [27] for our work, however the results and approach generalize to other tools.

To validate our work, we formally specify and verify a remote attestation protocol – known as the Privacy CA Protocol [1] – using commands from TPM version 1.2. Our objective is to capture an abstract specification from the TPM specification, validate it, and use it to verify the correctness of the Privacy CA Protocol. We are not making an argument for the protocol itself, we are merely verifying this protocol as a part of verifying the TPM.

In order to properly verify the TPM, we have designed an abstract state modeled off of the state of the TPM. We model commands that encapsulate how the state is changed. We define pre- and postconditions as per axiomatic semantics [20] for each command and using PVS's proof techniques, we are able to verify their validity. We use a state monad to pass modified state and command outputs across a sequence of commands to verify properties of protocols of TPM.

We have successfully verified about 40 commands from the TPM command set and several protocols dealing with key usage and sealing data. Our biggest achievement has come in verifying the Privacy CA Protocol [19]. Our CA Protocol steps through the role of the TPM in remote attestation and proves that the commands return what they are intended to return. Additional theorems verify invariants, postconditions, and detectability of various attacks. In the abstract model, we are focusing on continuing verification of the full TPM command set. We also plan to extend our work to include virtual TPMs.

Chapter 2

Background

Our work lies at the intersection of security, hardware, and formal methods. Although no mastery of any of these fields is required to understand our work, there are several areas that require some prior knowledge in order to understand what we are trying to accomplish with this effort. These areas include the TPM and related protocols, bisimulation, and our specification techniques.

2.1 Trusted Platform Module

The TPM [1] is a discrete hardware chip that provides cryptographic functions at the heart of establishing and maintaining a trusted computing infrastructure [7]. The TPM's functionality can be distilled into three major capabilities: (i) establishing, maintaining, and protecting a unique identifier; (ii) storing and securely reporting system measurements; and (iii) binding secrets to a specific platform.

TCG's specification of the TPM details 126 commands. Commands perform functionality ranging from updating flags to sending complex messages. We will model the state of the TPM and how commands change that state within PVS.

Although there are many parts to this, the most important to the TPM, and therefore also to our purposes, are the roots of trust which ensure that the TPM fits the requirements of being a trusted system. The Endorsement Key, used to identify the TPM is known as the root of trust for reporting. The Storage Root Key, a key for encrypting data, serves as the root of trust for storage. Platform Configuration Registers are used to store measurements, the root of trust for measurement, which are hashes of TPM configurations.

2.1.1 Keys

The *Endorsement Key* (EK) and *Storage Root Key* (SRK) are persistent asymmetric keys maintained by the TPM. EK uniquely identifies the TPM and EK^{-1} is maintained confidentially while EK encrypts secrets for use by TPM. EK^{-1} could theoretically sign TPM data, but is never used for this purpose to avoid unintended information aggregation. Instead, it provides a root-of-trust for reporting used in the attestation process.

As its name suggests, the Storage Root Key is a root key. The SRK provides a root for data encryption. The data being encrypted is often keys, known as *key wrapping*. A wrapped key is an asymmetric key pair whose private key is encrypted by another asymmetric key. For example, if we were to express an asymmetric key pair as (K, K^{-1}), *wrapping* key K with the SRK would result in $(K, \{K^{-1}\}_{SRK})$. Since the only way to get to K^{-1} is by decrypting using SRK^{-1}, the resulting wrapped key can be safely stored outside of the TPM. A wrapped key may only be installed and used if its wrapping key has been installed. In this way, chains of wrapped keys can be created, forming a tree structure with the SRK at its root. Using the SRK as the root of these chains binds all of the keys

to the specific TPM.

To provide platform authentication while maintaining the privacy of the EK, a TPM uses short-term *Attestation Identity Keys* (AIKs). AIKs differ from other keys in that they are only used to sign specific structures within the TPM. Using a certificate, a CA binds an AIK to a specific TPM without giving away the identity of that TPM [2, 9].

New keys within the TPM are typically created using the TPM command `TPM_CreateWrapKey`. These keys contain several attributes including a usage value, which tells the TPM whether the key is intended to be used as a storage key (such as the SRK), identity key (an AIK), signing key, or migration key; flags which indicate whether or not the key is migratable and volatile; and PCR information (discussed in the following section). As mentioned, it is possible for a wrapped key to be stored outside of the TPM. The EK and SRK are non-migratable, meaning that they are never allowed to be stored outside of the TPM.

2.1.2 Platform Configuration Registers

A *platform configuration register* (PCR) is a special purpose register for storing and extending hashes within the TPM. As its name implies, a PCR records a platform's configuration during boot or at run time. The TPM ensures the integrity of PCRs and uses a quote mechanism to deliver them with integrity to an external appraiser. Rather than being set to a specific value, PCRs are extended using the formula $pcr \parallel h = SHA1(pcr +\!\!+ h)$. These hashes – called *measurements* – are gathered in PCRs at various points during system operation, but the most common use is to ensure trusted boot. As each system component boots, images and data are hashed, and each hash is used to extend a PCR. The nature of extension

implies that at the conclusion of the boot process, the hashes in PCRs indicate whether the right parts were used in the right order during boot. Specifically, ideal PCR extension exhibits the property that $h_0 \parallel h_1 = h_1 \parallel h_0 \Leftrightarrow h_0 = h_1$. The only way to change a PCR value is with a platform reboot or by using the command `TPM_Extend`.

One unusual feature of PCRs is they can have one of two initial values. Resettable PCRs initialize to -1 (all 1s) while non-resettable PCRs reset to 0. Only `senter` can reset PCRs to 0, allowing an appraiser to tell immediately if `senter` was not called to initiate a measured boot process.

Using a combination of keys and PCRs, the TPM is able to *seal* data to a state. This is done by encrypting a blob of data, and only allowing it to be decrypted when the PCR values match chosen PCR values. A non-migratable key, K, is used to encrypt a data blob. This key and the chosen PCRs are then encrypted by a binding key, BK. The resulting structure can is represented by: $(\{Data\}_K, \{K, PCRs\}_{BK})$. Unsealing this data occurs when the current state PCR values match up with the PCRs encrypted by BK. This process of sealing data to the state is important for ensuring the PCRs values used are actually those stored in the TPM [1].

2.2 Protocols

As TPM operations often require multiple commands, it is an important part of TPM verification to verify command sequences, or protocols. If the TPM commands are correct, then sequencing them should result in correct protocols in addition to verifying the protocols themselves. In the actual hardware, protocols are executed simply by running the commands one after another. In our abstract

model, protocol execution is modeled using a state monad [26,33] to thread states through a sequence of TPM command executions.

The TPM is designed to be maximally flexible, meaning there is no set definition of what these protocols should be. Very few of the TPM commands do something that we want in isolation. For example, the command `TPM_CreateWrapKey` creates a key, and the command `TPM_LoadKey2` loads a key. There is no reason to create a key that will not be loaded, and it is impossible to load a key that has not been created. Therefore, it is obvious that these commands should be run sequentially. The most important protocol to our work thus far is the Privacy CA Remote Attestation protocol.

2.2.1 Remote Attestation

Remote Attestation using a TPM is the process of gathering PCRs and delivering them to an external appraiser in a trusted fashion [16]. By examining the reported contents of PCRs, the appraiser can determine whether it trusts the system described. Using hashes guarantees the appraiser only learns whether the right system is running and nothing more. Our remote attestation protocol uses a *Privacy Certificate Authority* (CA or Privacy CA) that produces an identity certificate verifying that an AIK public key belongs to a certain TPM using its EK [1,9]. The Privacy CA is so named because it protects the EK while assuring the AIK belongs to the right EK. This protocol is shown in Figure 2.1.

An AIK, wrapped by the SRK, is created using the TPM's `TPM_MakeIdentity` command and can only be used by the TPM that generated it. The command also returns a CA label digest identifying the CA certifying the AIK, and the public AIK signed with AIK^{-1}. The AIK signature tells us that the AIK came from the

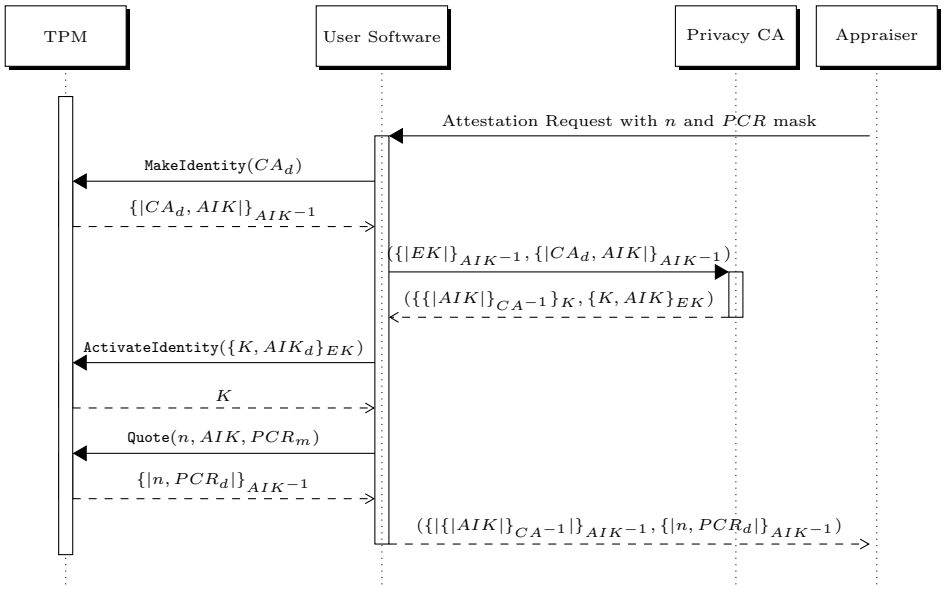

Figure 2.1. Sequence diagram for the Privacy CA protocol.

right TPM since the TPM that generated the AIK is the only entity with access to its private key. Using the public key embedded in the certificate, the CA can determine if the entire certificate did indeed come from the TPM associated with the AIK.

Although we are modeling the TPM, we also need to model the role of the Privacy CA. This interaction between the CA and the User is modeled by `CA_certify`. The CA returns a session key (identified as `K` within Figure 2.1) encrypted by the public EK associated with the TPM that claims to have requested the certificate. `TPM_ActivateIdentity` attempts to decrypt `K` using the TPM's EK^{-1} and releases if it decrypts successfully. Finally, we are able to use the AIK to sign PCR values using the TPM command `TPM_Quote` [1]. This quote is returned to the User who can then send back to the appraiser the information that it needs. The command `CPU_BuildQuoteFromMemory` simulates this final step generating for the

appraiser an evidence package of the form:

$$(\{|\{|AIK|\}_{CA^{-1}}|\}_{AIK^{-1}}, \{|n, PCR|\}_{AIK^{-1}}) \tag{2.1}$$

where: $\{|n, PCR|\}_{AIK^{-1}}$ is the nonce from the appraiser's request and desired PCR values; $\{|\{|AIK|\}_{CA^{-1}}|\}_{AIK^{-1}}$ is the certificate from a Privacy CA and public AIK; and both are signed by the AIK.

2.3 Bisimulation

The approach we take for verifying the TPM is to establish a *weak bisimulation* [30] relation between an abstract requirements model and a concrete model derived directly from the TPM specification as shown in Figure 2.2. Both the abstract and concrete models define transition systems in terms of system state and transitions over that system state.

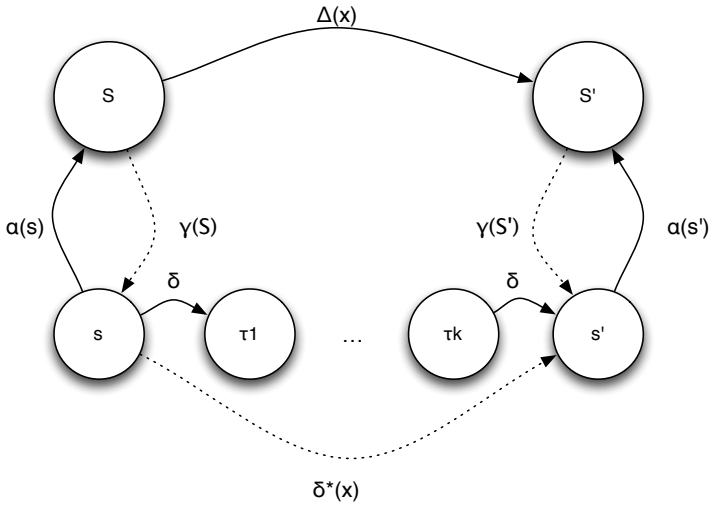

Figure 2.2. A weak bisimulation relation between an abstract transition system $Abs = (S, \Sigma, \Delta)$ and a concrete transition system $Con = (s, \sigma, \delta)$.

We say that $Abs = (S, \Sigma, \Delta)$ is an *abstract model* where S is a set of abstract states, Σ is a set of actions on states and input, and $\Delta : S \times \Sigma \rightarrow \Sigma$ is a transition on state and action. Similarly, we say that $Con = (s, \sigma, \delta)$ is a *concrete state* where s is a set of concrete states, σ is a set of actions on states and input, and $\delta : s \times \sigma \rightarrow \sigma$ is a transition function.

Obviously, since the TPM is a hardware chip, all data is stored and transmitted as bits. The concrete model of the TPM will encompass such details. The abstract model, appropriately is more abstract. It is significantly easier to verify properties about the TPM from an abstract level than to work with the concrete specification. Therefore, the abstract model was our starting point.

Eventually, we will relate the abstract and concrete models through an *abstraction function*, $\alpha : s \rightarrow S$, and *concretization function*, $\gamma : S \rightarrow 2^s$, which must form a Galois Connection. However, the work reported here focuses on the development, verification and validation of *Abs* for a TPM. In the abstract model defined following, `tpmAbsState` corresponds with S, `tpmAbsInput` corresponds with the abstract input, while `executeCom` and`outputCom` correspond to Δ.

2.4 Assumptions

In order to avoid complicated algorithms that have already been verified in their own right, we make several assumptions. We assume the hash function is injective, giving the property $SHA1(b_0) = SHA1(b_1) \Leftrightarrow b_0 = b_1$. Although in practice, hash collisions are possible, for our verification purposes, we assume that will never happen. Additionally, we abstract away the details of RSA keys, leaving only the assumption that all keys are unique and are unable to be hacked. We do not yet attempt to model specific attacks on the TPM, we assume a Dolev-Yao [15]

model for cryptography functions. Finally, though the TPM has a random number generator, we are not interested in verifying the algorithm itself, merely that the use of the results is effective. This leaves us with the assumption that the numbers generated are random.

2.5 Verification Technique

To do our verification of the TPM, we use the verification system PVS [27]. Developed by SRI, PVS is a specification language that supports an automated theorem prover. Fundamentally, it uses classical typed higher-order logic. Its built-in types include booleans and integers, and uninterpreted types may be introduced by the user. Type-constructors include functions, sets, tuples, records, and abstract data types. Predicate subtypes and dependent types can be used to introduce constraints. However, these constrained types may give rise to proof obligations (called type-correctness conditions or TCCs) during typechecking, most of which are discharged automatically by the theorem prover. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers.

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively by the user. The primitive inferences include propositional and quantifier rules, induction, rewriting, simplification using decision procedures, and data and predicate abstraction. We have designed most of our proofs to use the catch-all procedure `grind`, that repeatedly rewrites, simplifies, and applies decision procedures.

Chapter 3

System Model

Since no formal model of the TPM currently exists, our first task is to create one. We are working with the precisely written specification provided by the TCG for version 1.2 of the TPM. This section will show how we have abstractly modeled the data and commands of TPM from the TCG documents. All code written can be found in the Appendix at the end of this document.

3.1 TCG Specification

The TCG specification provides lists and tables of the data structures used by the TPM [1]. For example, Figure 3.1 shows the TCG defined structure of a *sealed data structure* (detailed explanation given in section 3.2). For each structure defined, TCG includes an informative section, a definition, and a table with details of each of the fields. Often these structures include fields (i.e. the field *payload*) that provide a level of detail that we do not need to model. Our abstract model seeks to capture only the functionality essential to the TPM commands, and therefore is able to ignore such details. It will, however, be necessary to reproduce

these elements for the concrete model.

9.3 TPM_SEALED_DATA

Start of informative comment

This structure contains confidential information related to sealed data, including the data itself.

End of informative comment

Definition

```
typedef struct tdTPM_SEALED_DATA {
   TPM_PAYLOAD_TYPE payload;
   TPM_SECRET authData;
   TPM_SECRET tpmProof;
   TPM_DIGEST storedDigest;
   UINT32 dataSize;
   [size_is(dataSize)] BYTE* data;
} TPM_SEALED_DATA;
```

Parameters

Type	Name	Description
TPM_PAYLOAD_TYPE	payload	This SHALL indicate the payload type of TPM_PT_SEAL
TPM_SECRET	authData	This SHALL be the AuthData data for this value
TPM_SECRET	tpmProof	This SHALL be a copy of TPM_PERMANENT_DATA -> tpmProof
TPM_DIGEST	storedDigest	This SHALL be a digest of the TPM_STORED_DATA structure, excluding the fields TPM_STORED_DATA -> encDataSize and TPM_STORED_DATA -> encData.
UINT32	dataSize	This SHALL be the size of the data parameter
BYTE*	data	This SHALL be the data to be sealed

Figure 3.1. Example TCG specification - `TPM_SEALED_DATA`.

Due to the size limitations of the TPM hardware, many commands include data on the size of the data it holds (i.e. *dataSize*). In our abstract model, we completely omit this size information since we do not model this detail. These constraints will also be included in the concrete model, but would provide an unnecessary level of detail in our abstract model.

From the table of Figure 3.1, we can see from the type given that the *data* field of the sealed data structure is a string of bytes. Taking a more in-depth look at the TCG specification of any of the remaining types in Figure 3.1 (`TPM_SECRET` and `TPM_DIGEST` for example), we see these are also structures constructed of bytes, as we would expect from a hardware implementation. We wish, however, to abstract away from the bit- and byte-level. Therefore, we discard implementation data

structures while preserving the semantics of its content.

3.2 Data Model

To preserve the semantics of the data structures of the TCG's specification, we define our own types for the TPM data structures. We model the same `TPM_SEALED_DATA` shown in Figure 3.1 as the PVS structure shown in Figure 3.2. For the most part, we keep the names of our PVS structures the same as those from the TCG specification. Notice that the *payload* and *dataSize* are excluded from our `tpmSealedData` structure as they are irrelevant to our model for reasons previously explained.

```
tpmSealedData(authData:(tpmSecret?),
              tpmProof:(tpmSecret?),
              storedDigest:(tpmDigest?),
              data:tpmData,
              crs:CRYPTOSTATUS) : tpmSealedData?
```

Figure 3.2. PVS abstraction of `TPM_SEALED_DATA`.

We have defined all of the data relevant to our abstraction in a recursive data type called `tpmData`, a portion of which is shown in Figure 3.3. Within this data type, we build structures out of other `tpmData` types and previously defined PVS data types.

The `tpmDigest` type is our representation of the `TPM_DIGEST`. To perform a hash of data, the TPM concatenates the values to be hashed and then takes the SHA1 hash of them, represented as $SHA1(d_0 +\!\!+ d_1 +\!\!+ ... +\!\!+ d_n)$. In our model, the `tpmDigest` is represented as a *list* of the elements that are to be concatenated and then hashed. For example, this same SHA1 hash is modeled as `tpmDigest(cons(d_0,cons(d_1...,cons(d_n,null))))`. This enables us to be

```
tpmData : DATATYPE
BEGIN
  tpmDigest(digest:list[tpmData],crs:CRYPTOSTATUS) : tpmDigest?
  tpmNonce(i:int) : tpmNonce?
  tpmSecret(i:int) : tpmSecret?
  tpmCompositeHash(dig:PCR_COMPOSITE) : tpmCompositeHash?
  tpmPCRInfoLong(locAtCreation:LOCALITY,locAtRelease:LOCALITY,
     creationPCRSelect:PCR_SELECTION,releasePCRSelect:PCR_SELECTION,
     digAtCreation:(tpmCompositeHash?),digAtRelease:(tpmCompositeHash?)) :
     tpmPCRInfoLong?
  tpmKey(key:KVAL,keyUsage:KEY_USAGE,keyFlags:KEY_FLAGS,
     PCRInfo:(tpmPCRInfoLong?),wrappingKey:KVAL,
     crs:CRYPTOSTATUS) : tpmKey?
  tpmSessKey(skey:KVAL,crs:CRYPTOSTATUS) : tpmSessKey?
  tpmQuote(digest:(tpmCompositeHash?),externalData:(tpmNonce?),
     crs:CRYPTOSTATUS) : tpmQuote?
  tpmStoredData(sealInfo:(tpmPCRInfoLong?),encData:(tpmSealedData?),
     crs:CRYPTOSTATUS) : tpmStoredData?
  tpmSealedData(authData:(tpmSecret?),tpmProof:(tpmSecret?),
     storedDigest:(tpmDigest?),data:tpmData,crs:CRYPTOSTATUS) :
     tpmSealedData?
  ...
END tpmData;
```

Figure 3.3. Abstract data type for TPM data.

able to see inside of the resulting hash value so that we have to model neither the actual SHA1 hash nor concatenation functions.

The `tpmNonce` and `tpmSecret` structures, while very similar in appearance, perform slightly different functions. A `tpmNonce` contains a random value that provides protection from replay attacks. We do not have a random number generator modeled, since we are not interested in verifying the validity of the random number generation algorithm. Therefore, we model the uniqueness properties using a monotonically increasing integer value. When we create nonces, we will be able to have a nonce count that we can verify never decreases, therefore never

using the same nonce value twice. The `tpmSecret` is used as a plaintext password to verify authorization. Since integers are easier to model than byte-strings, we have also given the `tpmSecret` an integer value.

To understand the `tpmCompositeHash` and `tpmPCRInfoLong` structures, we must look more in-depth at PCRs in general. Within the TPM, a `TPM_PCRVALUE` is a special `TPM_DIGEST` type that represents a PCR's contents. As explained in Section 2.1.2, PCRs can be reset to values of either 0 or -1, or they can be extended. We represent this in the PVS-defined `PCR` data type:

```
PCR : DATATYPE
BEGIN
  reset : reset?
  resetOne : resetOne?
  extend(pcr:PCR,hash:HV) : extend?
END PCR;
```

with `reset` corresponding to non-resettable PCRs (reset to 0), `resetOne` corresponding to resettable PCRs (reset to -1), and `extend` representing a sequence of values used to create the PCR value is maintained rather than calculate the hash.

The `PCR_SELECTION` is a list of PCR indexes (where order matters), and `PCRVALUES` is an array of `PCR`s. We have defined `PCR_COMPOSITE` as the record:

```
PCR_COMPOSITE : TYPE =  [# select : PCR_SELECTION
                         , pcrValue : PCRVALUES #];
```

where `select` indicates which PCR values are active, and `pcrValue` contains the values of all PCRs. Within the TPM, a `TPM_PCR_COMPOSITE` would really be a single blob concatenating only the values of the PCRs indicated by `select`, but for our model, it is sufficient to leave the data in this record type.

A `TPM_COMPOSITE_HASH` is a special `TPM_DIGEST` that takes the SHA1 digest of a `PCR_COMPOSITE`. Again, rather than calculate the hash, we leave the

PCR_COMPOSITE directly accessible from within the tpmCompositeHash structure.

When it comes to PCRs and digests, we are not interested in implementing actual hash algorithms, we instead want to capture at a higher level what the hash is trying encapsulate. We are interested to know if we have captured the correct elements in the correct order. Therefore, through our structures, we can focus on the elements and their order without modeling the details involved with concatenation of different data types or implementation of hash functions.

The tpmPCRInfoLong structure relates key wrapping or data sealing to a set of PCRs. The current locality, the selection of active PCRs, and a composite digest of the PCR values are captured as locAtCreation, creationPCRSelect, and digAtCreation, respectively, when when the blob is created. Additionally, the key or data may be bound to a specific selection of PCRs, releasePCRSelect. The digAtRelease is the digest of PCR indices and PCR values to verify when revealing sealed data or using a key that was wrapped to PCRs. Using the wrapped key or unsealing the data can only be done if a recalculated digestAtRelease matches the stored version. *Locality*, as used with locAtCreation and locAtRelease, is an additional security concept of the TPM indicating whether or not the TPM is in an acceptable state to run a specific command. Locality implements a kind of ring security model. It is not fully functional within our model, so for now, we will consider it a placeholder.

The tpmKey structure represents an asymmetric key pair with additional properties used by the TPM. These include its usage, associated flags, and PCR information for wrapping. Virtually all asymmetric keys used by the TPM are created as wrapped keys. Thus, a reference to the wrapping key is part of the tpmKey structure. Instead of modeling an actual RSA key, the type KVAL associated with

all keys is an integer value that uniquely identifies the key. We use a positive integer value for the public key, and the additive inverse of that same integer represents the private key. Integers were chosen as the representative value of keys for several reasons: associated public/private keys can easily be identified, it is easy to chose the next new key, PVS can easily induct over integers ensuring key uniqueness.

The remaining structures are defined by composing the previously explained `tpmData`. A `tpmSessKey` is simply a symmetric key. The `tpmQuote` structure, an important part of the Privacy CA Protocol, provides the means for the TPM to quote the current values of a list of PCRs. The `tpmStoredData` and `tpmSealedData` structures are used for sealing and unsealing data. Notice that the `tpmStoredData` includes a `tpmSealedData` structure, labeled as `encData`. However, this `encData` stores a digest of all data from the `tpmStoredData` excluding `encData` itself. This allows for the TPM to make sure that the data was not tampered with while performing sealing and unsealing operations.

Most elements of our `tpmData` data type include a tag that shows what cryptographic operations have been performed on data using the `CRYPTOSTATUS` type. These functions include encryption, signing, and sealing. For example, a symmetric key identified with `k:KVAL` and signed with the private key of `idKey:(tpmKey?)` is expressed as: $\texttt{tpmSessKey}(\texttt{k}, \texttt{signed}(\texttt{private}(\texttt{idKey}), \texttt{clear}))$.

3.3 Abstract State

The abstract state is modeled after the TPM's various state structures that manage the internal data stored by the TPM. The TPM manages state by maintaining several data fields and flags. We use a PVS record structure, referred to

as tpmAbsState and shown in Figure 3.4, to maintain an abstract view of this state as well as the memory associated with the environment where the TPM is being run. The elements most relevant to our verification include srk, ek, pcrs, and memory.

```
tpmAbsState : TYPE =
     [# restore : restoreStateData,
        memory : mem,
        srk : (tpmKey?),
        ek : (tpmKey?),
        keyGenCnt : K,
        keys : KEYSET,
        pcrs : PCRVALUES,
        locality : LOCALITY,
        permFlags : PermFlags,
        permData : PermData,
        stanyFlags : StanyFlags,
        stanyData : StanyData,
        stclearFlags : StclearFlags,
        stclearData : StclearData
     #];
```

Figure 3.4. Abstract TPM and system state record data structure.

The memory is not part of the actual TPM, but represents the memory used by the TPM's environment for storing values. This is necessary for our model due to our method of command sequencing discussed in section 3.5.

The srk and the ek represent the asymmetric keys SRK and EK used by the TPM as roots of trust previously discussed in section 2.1.1. keyGenCnt is used to simulate key generation. Each time a key is created, keyGenCnt is incremented, providing a unique integer to be assigned to each key. These keys, upon being loaded into the TPM are added to the keys set. The pcrs field of the state stores all of the current PCRs values.

The flag and data fields are broken down into the different ways the TPM

resets data. Upon startup, the TPM specifies what type of startup is occuring - the TPM is starting from a clean state (`ST_CLEAR`); the TPM is starting from a saved state (`ST_STATE`); or the TPM is to startup and set the deactivated flag to true (`ST_DEACTIVATED`). The `permFlags` and `permData` store permanent data that is not affected by any startup command; `stanyFlags` and `stanyData` are reset upon any of the three startup commands; and `stclearFlags` and `stclearData` are reset on each `ST_CLEAR` command. The `ek` and `srk` are part of `permData` within the TPM. `pcrs` fall under `stclearData`.

3.4 Command Definitions

The TCG specification describes TPM commands by providing a table of command inputs, a table of command outputs, and a step-by-step description of what the command does upon execution. We model this by defining a state transformation over the TPM state.

3.4.1 Command Inputs

Figure 3.5 shows the table of input values for the command `TPM_Extend`, the command used to extend a PCR. Each TPM command has such a table including the type, name, and description of all of the input parameters.

To translate these tables into our abstract model, we extract the relevant data and pass that data as parameters to the command. The `TPM_Extend` command is translated as: $\texttt{ABS\_Extend}(\texttt{pcrNum} : \texttt{PCRINDEX}, \texttt{d} : \texttt{HV}) : \texttt{ABS\_Extend}?$, where `d` corresponds to *inDigest* and `pcrNum` corresponds to the extended PCR number.

Each `TPM_`*Command* implemented has a corresponding `ABS_`*Command* within the `tpmAbsInput` data structure, as shown in Figure 3.6. Within the `tpmAbsInput`

16.1 TPM_Extend

Start of informative comment:

This adds a new measurement to a PCR

End of informative comment.

Incoming Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RQU_COMMAND
2	4			UINT32	paramSize	Total number of input bytes including paramSize and tag
3	4	1S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: TPM_ORD_Extend.
4	4	2S	4	TPM_PCRINDEX	pcrNum	The PCR to be updated.
5	20	3S	20	TPM_DIGEST	inDigest	The 160 bit value representing the event to be recorded.

Figure 3.5. Example of command input - `TPM_Extend`.

data structure, the arguments to each command are abstract representations of the actual TPM data formats, which typically come from the `tpmData` data type. This is appropriate for an abstract model where we are capturing functionality, not implementation. Some bit-level details are abstracted away when they do not contribute to verifying the basic functionality of the device.

Including all abstracted commands within the `tpmAbsInput` datatype gives us an induction principle for the command set that is automatically usable by PVS to quantify over all possible TPM commands and inputs. This allows us to prove things about state fields that are invariant over all commands (as explained in section 5.2).

3.4.2 Abstract Outputs

Figure 3.7 shows the table of output values for the `TPM_Extend` command. All TPM commands are accompanied by such a table which includes the type, name, and description of each of the output parameters. Each TPM command returns an output, sometimes just to return a message that the command was successfully

```
tpmAbsInput : DATATYPE
BEGIN
  ABS_Init : ABS_Init?
  ABS_SaveState : ABS_SaveState?
  ABS_Startup(startupType:TPM_STARTUP_TYPE) : ABS_Startup?
  ABS_TakeOwnership(srk:(tpmKey?)) : ABS_TakeOwnership?
  ABS_Seal(k:(tpmKey?),pcrInfo:(tpmPCRInfoLong?),inData:tpmData) :
       ABS_Seal?
  ABS_Unseal(parent:(tpmKey?),inData:(tpmStoredData?)) : ABS_Unseal?
  ABS_UnBind(key:(tpmKey?),inData:(tpmBoundData?)) : ABS_UnBind?
  ABS_CreateWrapKey(parentH,keyInfo:(tpmKey?)) : ABS_CreateWrapKey?
  ABS_LoadKey2(parent,inKey:(tpmKey?)): ABS_LoadKey2?
  ABS_MakeIdentity(CADigest:(tpmDigest?),idKey:(tpmKey?)) : ABS_MakeIdentity?
  ABS_ActivateIdentity(aik:(tpmKey?),b:(activateIdentityBlob?)) :
       ABS_ActivateIdentity?
  ABS_Extend(pcrNum:PCRINDEX,d:HV) : ABS_Extend?
  ABS_Quote(aik:(tpmKey?),nonce:(tpmNonce?),pm:PCR_SELECTION) :
       ABS_Quote?
  ABS_save(i:nat,v:tpmAbsOutput) : ABS_save?
  ABS_read(i:nat) : ABS_read?
  ABS_certify(aik:(tpmKey?),certReq:(tpmIdContents?)) : ABS_certify?
  ...
END tpmAbsInput;
```

Figure 3.6. Representative elements from the input data type.

run.

16.1 TPM_Extend

Outgoing Operands and Sizes

PARAM		HMAC		Type	Name	Description
#	SZ	#	SZ			
1	2			TPM_TAG	tag	TPM_TAG_RSP_COMMAND
2	4			UINT32	paramSize	Total number of output bytes including paramSize and tag
3	4	1S	4	TPM_RESULT	returnCode	The return code of the operation.
		2S	4	TPM_COMMAND_CODE	ordinal	Command ordinal: TPM_ORD_Extend.
4	20	3S	20	TPM_PCRVALUE	outDigest	The PCR value after execution of the command.

Figure 3.7. Example of command output - TPM_Extend.

Like inputs to the TPM, outputs are modeled abstractly using an algebraic

type. Again we avoid the complexity of bit-level representations specified in the TPM standard in favor of an abstract representation that captures the essence of TPM functionality. Figure 3.8 shows the representation of this type.

```
tpmAbsOutput : DATATYPE
BEGIN
  OUT_Init(m:ReturnCode) : OUT_Init?
  OUT_SaveState(m:ReturnCode) : OUT_SaveState?
  OUT_Startup(m:ReturnCode) : OUT_Startup?
  OUT_TakeOwnership(srk:(tpmKey?),m:ReturnCode) : OUT_TakeOwnership?
  OUT_Seal(sealedData:(tpmStoredData?),m:ReturnCode) : OUT_Seal?
  OUT_Unseal(secret:tpmData,m:ReturnCode) : OUT_Unseal?
  OUT_UnBind(outData:tpmData,m:ReturnCode) : OUT_UnBind?
  OUT_CreateWrapKey(wrappedKey:(tpmKey?),m:ReturnCode) :
       OUT_CreateWrapKey?
  OUT_LoadKey2(inkeyHandle:(tpmKey?),m:ReturnCode) : OUT_LoadKey2?
  OUT_MakeIdentity(idKey:(tpmKey?),idBinding:(tpmIdContents?),
       m:ReturnCode) : OUT_MakeIdentity?
  OUT_ActivateIdentity(symmKey:(tpmSessKey?),m:ReturnCode) :
       OUT_ActivateIdentity?
  OUT_Extend(outDigest:PCR,m:ReturnCode) : OUT_Extend?
  OUT_Quote(pcrData:list[PCR],sig:(tpmQuote?),m:ReturnCode) :
       OUT_Quote?
  OUT_FullQuote(quote:(tpmQuote?),idBind:(tpmIdContents?),
       m:cpuReturn) : OUT_FullQuote?
  OUT_Certify(k:(tpmKey?),dat:(tpmAsymCAContents?),m:cpuReturn) :
       OUT_Certify?
  OUT_Error(m:ReturnCode) : OUT_Error?
  OUT_CPUError(m:cpuReturn) : OUT_CPUError?
  ...
END tpmAbsOutput;
```

Figure 3.8. Representative elements from the output data type.

The `tpmAbsOutput` constructs allow for each command to return the correct output parameters as well as a return code. These return codes either indicate success or a non-fatal error. Fatal errors from TPM commands are generated using the `OUT_Error` construct, while non-TPM-related fatal errors are generated

using `OUT_CPUError`. Since we are modeling hardware, fatal errors are obviously terminating conditions. However, due to our command sequencing method (discussed in section sec:seq), these errors are merely propagated through to the end of the command sequence. Therefore, it is important to note that these structures are not part of the actual TPM, but are essential to our model.

3.4.3 Abstract Command Execution

Our technique for specifying TPM command execution is to define state transition and output functions in the canonical fashion for transition systems. Specifically, we define the `executeCom` function as a transition from `tpmAbsState` (Figure 3.4) and `tpmAbsInput` (Figure 3.6) to `tpmAbsState`:

```
executeCom : tpmAbsState → tpmInput → tpmAbsState
```

and the function `outputCom` to transform `tmpAbsState` and `tpmAbsInput` into a `tpmAbsOutput` (Figure 3.8) value:

```
outputCom : tpmAbsState → tpmAbsInput → tpmAbsOutput
```

Given $s : tpmAbsState$ and $c : tpmAbsInput$, the output-state pair resulting from executing c is defined as:

(`outputCom`(s,c),`executeCom`(s,c))

The `executeCom` and `outputCom` functions are defined by cases over `tpmAbsInput` (Figures 3.9 and 3.10 respectively). Specifically, for each command in `tpmAbsInput`, a function is defined for generating the next state and for generating output. These commands are named within the specification using the suffix `State` and `Out` respectively for easy identification.

```
executeCom(s:tpmAbsState,c:tpmAbsInput) : tpmAbsState =
  CASES c OF
    ABS_Init : tpmPostInit,
    ABS_SaveState : saveState(s),
    ABS_TakeOwnership(srk) : takeOwnershipState(s,srk),
    ABS_CreateWrapKey(p,k) : createWrapKeyState(s,p,k),
    ABS_LoadKey2(p,k): loadKey2State(s,p,k),
    ABS_MakeIdentity(d,k) : makeIdentityState(s,d,k),
    ABS_Extend(n,d) : extendState(s,n,d),
    ABS_certify(k,cr) : certState(s,k,cr),
    ...
    ELSE s
  ENDCASES;
```

Figure 3.9. The `executeCom` function.

Since all TPM commands return at least a success or error message, all abstract commands generate output, but not all commands modify state. In instances where the state is not modified, the `CASES` construct used to assemble the functions defaults to not modifying the state.

```
outputCom(s:tpmAbsState,c:tpmAbsInput) : tpmAbsOutput =
  CASES c OF
    ABS_TakeOwnership(srk) : takeOwnershipOut(s,srk),
    ABS_Seal(k,p,data) : sealOut(s,k,p,data),
    ABS_Unseal(d,k) : unsealOut(s,d,k),
    ABS_UnBind(k,d) : unBindOut(s,k,d),
    ABS_CreateWrapKey(parent,k) : createWrapKeyOut(s,parent,k),
    ABS_LoadKey2(p,k) : loadKey2Out(s,p,k),
    ABS_MakeIdentity(d,k) : makeIdentityOut(s,d,k),
    ABS_ActivateIdentity(i,b) : activateIdentityOut(s,i,b),
    ABS_Extend(n,d) : extendOut(s,n,d),
    ABS_Quote(k,n,pm) : quoteOut(s,k,n,pm),
    ABS_certify(aik,cr) : certOut(s,aik,cr),
    ...
    ELSE OUT_Error(TPM_SUCCESS)
  ENDCASES;
```

Figure 3.10. The `outputCom` function.

The TCG specification gives a list of actions for the TPM to perform upon execution. Consider again the `TPM_Extend` command. Figure 3.11 shows the actions the TPM is required to execute upon calling this command.

16.1 TPM_Extend

Actions

1. Validate that pcrNum represents a legal PCR number. On error, return TPM_BADINDEX.
2. Map L1 to TPM_STANY_FLAGS -> localityModifier
3. Map P1 to TPM_PERMANENT_DATA -> pcrAttrib [pcrNum]. pcrExtendLocal
4. If, for the value of L1, the corresponding bit is not set in the bit map P1, return TPM_BAD_LOCALITY
5. Create c1 by concatenating (TPM_STCLEAR_DATA -> PCR[pcrNum] || inDigest). This takes the current PCR value and concatenates the inDigest parameter.
6. Create h1 by performing a SHA-1 digest of c1.
7. Store h1 to TPM_STCLEAR_DATA -> PCR[pcrNum]
8. If TPM_PERMANENT_FLAGS -> disable is TRUE or TPM_STCLEAR_FLAGS -> deactivated is TRUE
   a. Set outDigest to 20 bytes of 0x00
9. Else
   a. Set outDigest to h1

Figure 3.11. Example of command actions- `TPM_Extend`.

At its core, the command `TPM_Extend` extends a specific PCR value. However, the TPM must first perform some checks to make sure that the input values are correct and that the TPM is in a state where performing this command is acceptable. From Figure 3.11, steps 1.-4. perform these checks, while steps 5.-9. carry out the functionality of the command.

The function `extendState` defines only how the `TPM_Extend` command modifies the state of the TPM:

```
extendState(s:tpmAbsState,pcrNum:PCRINDEX,inDigest:HV) : tpmAbsState =
  LET L1=s‘stanyFlags‘localityModifier,
      P1=pcrExtendLocal(s‘permData‘pcrAttrib(pcrNum)) IN
  IF (0<=pcrNum<=23) AND member(L1,P1)
    THEN s WITH [‘pcrs := pcrsExtend(s‘pcrs,pcrNum,inDigest)]
    ELSE s
```

```
  ENDIF
```

If the `TPM_Extend` command will throw an error – if `pcrNum` is not a legal PCR number, or if the `localityModifier` flag is not set – then the state is not modified. Otherwise, the value of PCR *pcrNum* is extended within the TPM state by *inDigest* using the helper function `pcrsExtend`.

The function `extendOut`, shown below, defines the TPM output generated by the `TPM_Extend` command as defined in the TCG specification, shown in Figure 3.11.

```
extendOut(s:tpmAbsState,pcrNum:PCRINDEX,inDigest:HV) : tpmAbsOutput =
  LET L1=s‘stanyFlags‘localityModifier,
      P1=pcrExtendLocal(s‘permData‘pcrAttrib(pcrNum)),
      H1=pcrsExtend(s‘pcrs,pcrNum,inDigest) IN
  IF pcrNum > 23 OR pcrNum < 0
    THEN OUT_Error(TPM_BADINDEX)
  ELSIF not member(L1,P1)
    THEN OUT_Error(TPM_BAD_LOCALITY)
  ELSIF s‘permFlags‘disable OR s‘stclearFlags‘deactivated
    THEN OUT_Extend(reset,TPM_SUCCESS)
    ELSE OUT_Extend(extend(s‘pcrs(pcrNum),inDigest),TPM_SUCCESS)
  ENDIF
```

The `extendOut` function first checks the error cases, returning an error message as an `OUT_Error`. Otherwise, the function returns `OUT_Extend` where the appropriate single PCR value is returned along with a return message indicating a successfully executed command.

The *command*`Out` and *command*`State` (if applicable) are then combined to make the entire `TPM_`*Command* using either `modifyOutput` if updating state and producing an output, or `output` if only producing an output. `TPM_Extend` is shown below, where `State : (tpmAbsOutput,tpmAbsState)`:

```
TPM_Extend(n:PCRINDEX,h:HV):State =
```

```
modifyOutput(
   (LAMBDA (s:tpmAbsState):executeCom(s,ABS_Extend(n,h))),
   (LAMBDA (s:tpmAbsState):outputCom(s,ABS_Extend(n,h))));
```

3.5 Sequencing Command Execution

In order to fully model a protocol, we must sequence commands in a manner like assembly commands in a traditional microprocessor. Sequencing of TPM commands is a matter of using the output state from one command as the input to the next command. The classical mechanism for doing this involves executing a command and manually feeding its resulting state to the next command in sequence. Using a `LET` form, executing `i;i'` would look like the following:

```
LET (o',s') = (outputCom(s,i),executeCom(s,i)) IN
                     (outputCom(s',i'),executeCom(s',i'))
```

We choose to use an alternative approach that uses a state monad [26, 33] to model sequential execution. The state monad threads the state through sequential execution in the background. The result is a modeling and execution pattern that closely resembles the execution pattern of TPM commands.

To understand the state monad, let us first define a simple data type, `State`, having a single field called `state` that holds a function from an abstract state to an abstract state, abstract output pair:

```
State : DATATYPE
BEGIN
    state(runState:[tpmAbsState->[tpmAbsOutput,tpmAbsState]]):state?
END State;
```

Given s, a value of type `tpmAbsState`, and m, a `State`, the application `runState(m)(s)` will result in a `tpmAbsOutput`, `tpmAbsState` pair. This is pre-

cisely the output expected. Note that the use of `State` and `state` in this definition is somewhat misleading. Neither is actually a state, but a state monad that given a state will generate a new state. The data type should be viewed as a kind of state generation or next-state function, not a single state. This is in fact the PVS data type that we use to represent State.

Two functions must be defined for any instance of a monad – return and bind (`>>=`). We will also define sequence (`>>`) command that is a special case of bind commonly defined in most state monad implementations.

First we define `return` whose form is:

```
return(x:tpmAbsOutput):State =
    state(LAMBDA (s:tpmAbsState) : (x,s))
```

where `return` lifts a member of `tmpAbsState` into `State` – given a `tpmAbsState`, it returns a `State` that when run produces the original `tpmAbsState`. It is important to recognize that the resulting value is of type `State`. This is not a TPM state, but a state monad that can produce a state. If we extract the `runState` function and apply it to a `tpmAbsState` value we will get the output, state pair that we input. Specifically, `runState(return(a))(st)=(a,st)`.

The second function defined for all monads is bind, typically represented by the infix operator `>>=`. The bind operation takes a monad and a function from `tpmAbsOutput` to a monad and produces a new monad. The implementation is:

```
>>= (m:State,f:[A->State]):State =
  state(LAMBDA(s0:S):
    LET (a,s1) = runState(m)(s0) IN
      runState(f(a))(s1));
```

Keep in mind that bind does not produce a state. Instead it produces a state monad that given a state, the parameter `s0` in the above implementation, will

produce an output, state pair.

The biggest clue to the behavior of bind comes in the `LET` form where `(a,s1)` is bound to running `m` – the first argument to bind – on state `s0`. `a` is bound to the output and `s1` to the state resulting from running `m` on `s0`. If we were doing this outside the monad, we would refer to this as the intermediate state between the two executions.

The result of the bind is `runState(f(a))(s1)`. First consider `runState(f(a))`. Looking at the signature, `f(a)` is a mapping from something of type `A` to a monad of type `State`. What we get, is a next state monad with the previous output bound to `a` – the previous output is available to the calculation of the next state. `runState` pulls the state function out of the new state monad and evaluates that function with `s1`, the intermediate state. So, the state is threaded through the evaluation with the user providing only `s0`, the initial state.

Another common operation is a specialization of bind that we refer to as *sequence* (`>>`) is shown here:

```
>> (m:State,f:State):State =
  state(LAMBDA(s0:S):
    LET (a,s1) = runState(m)(s0) IN
      runState(f)(s1));
```

Sequence works exactly like bind except that the previous output is ignored. As a simple example of what we are after, consider the TPM command sequence for modeling boot. Specifically, initialization followed by startup, *senter*, and *sinit* in sequence:

```
TPM_Init
>> TPM_Startup(TPM_ST_CLEAR)
>> CPU_senter
>> CPU_sinit
```

Chapter 4

Command Implementation

The TCG specification details functionality of 126 TPM commands. However, counting only those necessary for providing functionality for the TPM (some are depreciated, others optional), we must model 97 commands. We presently have 40 of these implemented listed in Table 4.1. Initially, we modeled the commands that make up the main functionality of the TPM and those commands necessary to verify the attestation protocol. Most of the commands that we have left to implement fall in the categories of authorization, delegation, and cryptography.

4.1 Important Commands

Among the most important commands to the TPM is `TPM_TakeOwnership`. The principle task of this command is to create the SRK, necessary for TPM since it is the base key in the key management tree [7, 23]. The `TakeOwnership` command functions check several state fields and flags, such as if there is already an owner and the validity of the EK, before establishing the SRK.

An important functionality of the TPM is the ability to use asymmetric keys to

Category	*Commands*	
Admin Startup and State	TPM_Init	TPM_Startup
	TPM_SaveState	
Admin Opt-in	TPM_SetOwnerInstall	TPM_OwnerSetDisable
	TPM_PhysicalEnable	TPM_PhysicalDisable
	TPM_PhysicalSetDeactivated	TPM_SetTempDeactivated
	TPM_SetOperatorAuth	
Admin Ownership Commands	TPM_TakeOwnership	TPM_OwnerClear
	TPM_ForceClear	TPM_DisableOwnerClear
	TPM_DisableForceClear	TSC_PhysicalPresence
	TSC_ResetEstablishmentBit	
Protected Storage Commands	TPM_Seal	TPM_Unseal
	TPM_UnBind	TPM_CreateWrapKey
	TPM_LoadKey2	TPM_GetPubKey
Migration Commands	TPM_CreateMigrationBlob	TPM_ConvertMigrationBlob
	TPM_AuthorizeMigrationKey	TPM_MigrateKey
Cryptographic Commands	TPM_Sign	
Endorsement Key Handling	TPM_CreateEndorsementKeyPair	TPM_CreateRevocableEK
	TPM_RevokeTrust	TPM_ReadPubek
	TPM_OwnerReadInternalPub	
Identity Creation and Activation	TPM_MakeIdentity	TPM_ActivateIdentity
Integrity Collection and Reporting	TPM_Extend	TPM_PcrRead
	TPM_Quote	TPM_PCR_Reset
Non-TPM Commands	Tspi_Data_Bind	CPU_senter
	CPU_sinit	CPU_BuildQuoteFromMem
	CPU_saveOutput	CPU_read
	CA_certify	

Table 4.1. Implemented Commands.

seal and bind data. Commands such as `TPM_CreateWrapKey` and `TPM_LoadKey2` create these asymmetric keys and enable them for use within the TPM. As mentioned in Section 3.2, public keys are modeled as `KVAL`s, and their private counterpart is the corresponding additive inverse. Therefore, creating a new key is mostly a matter of getting the next integer value. Within the TPM, when a key is created, the private part of the key is wrapped by a key. Each new key must be wrapped with either the SRK or a key who can trace its roots to the SRK, therefore creating the tree structure. Instead of modeling this encryption in our model, we simply pass the identity of the wrapping key as a parameter of the

tpmKey structure that is output after running the TPM_CreateWrapKey command. Current PCR info from the state is also stored with the key.

Before a key can be used to perform any actions within the TPM, it must be loaded. Since actually loading a key within the TPM consists of loading the key data into the internal memory, we model this by adding the key to be loaded to a list of KVALs maintained by system state.

After the key has been loaded, it can be used to seal and unseal data. The TPM_Seal command performs encryption on the data input parameter. Additionally, PCR values may be specified to be used in the seal [7]. In our model, this encryption is modeled within the CRYPTOSTATUS of the tpmSealedData structure, which is part of the tpmStoredData that is output from the TPM_Seal command. The command TPM_Unseal performs the corresponding decryption of the data.

We have already shown that the TPM_Extend command extends a PCR value. The TPM_Quote command outputs a quote, which consists of the PCR values signed by a tpmKey. This is so the state of the TPM can be sent to a third party. We model this third party using the command CPU_BuildQuoteFromMem.

The key used to sign quotes is an Attestation Identity Key. Identity keys are used in lieu of an EK for security reasons to identify a TPM, and must be certified as belonging to the tpm. The TPM_MakeIdentity command creates an identity key, which is then sent with the certified public EK to the Privacy CA [29]. Details of what specifically occurs within the Privacy CA are not relevant to the TPM. However, we do know that the command TPM_ActivateIdentity follows TPM_MakeIdentity, so looking at the inputs necessary to run that command, we know what the Privacy CA should output. This is modeled within the CA_Certify command. CA_Certify signs a certificate for the AIK and en-

crypts the certificate with a new session key. This new session key along with the public AIK is encrypted with the EK and sent to `TPM_ActivateIdentity`. `TPM_ActivateIdentity` decrypts this and releases the session key.

Two remaining commands that are not a part of the TPM, but are vital to our model are `CPU_saveOutput` and `CPU_read`. These commands allow us to store outputs of the commands in `memory` within the `tpmAbsState`. As previously mentioned, the `memory` is not part of the actual TPM, but is necessary for our model due to our use of the state monad for command sequencing. These commands allow us to both write to and read from specific memory locations (stored as natural numbers).

4.2 Command Predicates

Many commands require precondition checks to ensure either that the TPM is in an acceptable state to run the command or that the parameters passed to the command are correct. Since there are often multiple fields to verify, along with making the *command*`State` and *command*`Out` functions for these commands we additionally make a precondition function, *command*`?:bool`, function that returns a boolean value indicating if the precondition has been met. The practice of defining preconditions comes from axiomatic semantics [20] These predicates can then be used within both the *command*`State` and *command*`Out` functions as well as the proofs using these functions. The need for these command predicates varies by the complexity of specific commands.

Returning to the `TPM_Extend` example, from the command actions specified by the TCG in Figure 3.11, we can see that Action 1 should return the error message `TPM_BADINDEX` if the *pcrNum* input is an invalid value. Additionally,

Action 4 returns a `TPM_BAD_LOCALITY` error if the locality stored within the state is incorrect. Therefore, the `extend?` command will return true only if these error cases are not reached:

```
extend?(s:tpmAbsState,pcrNum:PCRINDEX,inDigest:HV) : bool =
  LET L1 = s`stanyFlags`localityModifier,
      P1 = pcrExtendLocal(s`permData`pcrAttrib(pcrNum))  IN
  (0<=pcrNum<=23) AND member(L1,P1)
```

To use this predicate, we can simplify the function `extendState` (originally introduced in Section 3.4.3) to update the state only if the predicate returns true:

```
extendState(s:tpmAbsState,pcrNum:PCRINDEX,inDigest:HV) : tpmAbsState =
  IF extend?(s,pcrNum,inDigest)
    THEN s WITH [`pcrs := pcrsExtend(s`pcrs,pcrNum,inDigest)]
    ELSE s
  ENDIF
```

An example of use of preconditions within proofs is shown in Section 5.4.1. As previously mentioned, the command set implemented thus far minimally provides us with the ability to model a full remote attestation protocol. Some aspects of individual commands have yet to be implemented – for example, a level of authorization using a concept called *authData* has been omitted since it is a further layer of TPM security and not a necessity for remote attestation even though it is an aspect of most commands. However, the functionality that has been modeled is sufficient to capture the role of each command.

Chapter 5

Verification Results

To verify our requirements model we verify individual commands with respect to their postconditions and invariants. To provide a degree of validation, we use those commands to model protocols and verify execution results. While some aspects of attacks are considered, we focus first and foremost on the correctness of the protocol implementation.

5.1 Individual Commands

For each TPM command, we define postconditions and verify that our abstract specifications meet those properties. For each command, we must show that given any value for all parameters of a command, running that command produces an output-state pair that satisfies the postcondition while not altering any invariant. Returning to our example of the `TPM_Extend` command, we verify the postconditions of our command execution with the theorem shown in Figure 5.1.

The `LET` form runs the command starting from any state in the predicate subtype `(afterStartup?)`. This predicate ensures that the start state, called

```
extend_post : THEOREM
 FORALL (state:(afterStartup?),h:HV,n:PCRINDEX) :
  LET (a,s) = runState(
                TPM_Extend(n,h))
                (state) IN
    LET L1=s‘stanyFlags‘localityModifier,
        P1=pcrExtendLocal(state‘permData‘pcrAttrib(n)),
        H1=pcrsExtend(state‘pcrs,n,h) IN
  IF extend?(state,n,h)
    THEN IF state‘permFlags‘disable OR state‘stclearFlags‘deactivated
           THEN a=OUT_Extend(reset,TPM_SUCCESS) AND
                s=state WITH [‘pcrs:=H1]
           ELSE a=OUT_Extend(extend(state‘pcrs(n),h),TPM_SUCCESS) AND
                s=state WITH [‘pcrs:=H1]
         ENDIF
  ELSIF n>23 OR n<0
    THEN a=OUT_Error(TPM_BADINDEX) AND
         s=state
  ELSIF not member(L1,P1)
    THEN a=OUT_Error(TPM_BAD_LOCALITY) AND
         s=state
    ELSE a=OUT_Error(TPM_SUCCESS) AND
         s=state
  ENDIF;
```

Figure 5.1. Verifying postconditions of `TPM_Extend`.

`state` is any valid `tpmAbsState` after the initialization commands have been run. The pair `(a,s)` is the resulting output and state respectively. The remainder of the theorem defines conditions on proper execution of the `TPM_Extend` command including both error and success cases.

In some situations it is necessary to include an `ELSE` clause where there is no possible output. Notice in Figure 5.1, there is a chance for the output to be `OUT_Error(TPM_SUCCESS)`. Recall that `OUT_Error` is not a part of the TPM specifications, but merely our version of fatal error handling. Therefore, by our own specifications, the concept of a "successful" error being thrown is not possible,

yet in some cases where we are forced to give an output, we use this. For commands where this is a possible issue, we include another theorem that shows that this case is unreachable. An example of this can be seen in Figure 5.2.

```
extend_post2 : THEOREM
 FORALL (state:(afterStartup?),h:HV,n:PCRINDEX) :
  LET (a,s) = runState(
                TPM_Extend(n,h))
                (state) IN
    LET L1=s`stanyFlags`localityModifier,
        P1=pcrExtendLocal(state`permData`pcrAttrib(n)),
        H1=pcrsExtend(state`pcrs,n,h) IN
  not extend?(state,n,h) =>
        not a=OUT_Error(TPM_SUCCESS)
```

Figure 5.2. Verifying postconditions of `TPM_Extend`.

For the `TPM_Extend` command, if the inputs given do not satisfy the `extend?` predicate, then an error message should be output. With use of a lemma that different return codes have different values, we are able to prove that the output of running `TPM_Extend` will never actually be `OUT_Error(TPM_SUCCESS)`.

Single-command-based proofs are straightforward and were designed to be able to be solved by using the catch-all strategy `grind` which repeatedly rewrites, simplifies, and applies decision procedures. Occasionally, we must use the strategy `decompose-equality`, which breaks down an equality where the terms are of function, record, tuple, or a datatype constructor type.

5.2 Invariants

In addition to defining and verifying postconditions of each TPM command, we also verify the invariance of properties that we wish to hold over command execution. Invariants in the model take two forms - those that are explicitly

defined and those that are captured in the abstract state type definitions. As was previously mentioned, the only way to change a PCR value is by rebooting the platform or using the `TPM_Extend` command. We can prove that this property holds for our model using the following theorem:

```
pcrs_unchanged: THEOREM
 FORALL (s:tpmAbsState,c:tpmAbsInput) :
  not(ABS_Startup?(c) OR ABS_Init?(c) OR
      ABS_senter?(c) OR ABS_sinit?(c) OR
      ABS_PCR_Reset?(c) OR ABS_Extend?(c)) =>
  pcrs(s) = pcrs(executeCom(s,c));
```

The theorem shows that with the exception of the commands `ABS_Startup`, `ABS_Init`, `ABS_senter`, `ABS_sinit`, `ABS_PCR_Reset`, and `ABS_Extend`, the value of the `pcrs` within the state, `s`, is the same before and after execution.

Notice that we have said that `TPM_Extend` is the only command that can modify a PCR value, and yet we have mentioned several other commands that allow these values to change. The `ABS_Startup` and `ABS_Init` commands set up standard initial states following the startup command and hardware initialization, respectively. These commands reset all fields within `tpmAbsState` and are therefore exceptions to all invariants. `ABS_PCR_Reset` is similar, except instead of resetting all state fields, merely the PCR values are reset. The remaining commands are the startup after reboot commands, which define initial states, and therefore are also an exception.

Table 5.1 shows which TPM commands are allowed to change each of the state fields. The field remains unchanged over all commands not listed.

It is interesting to note that while postconditions theorems are associated with individual commands, the invariants are typically proven over all commands simultaneously using the induction principle associated with the `tpmAbsInput`

State Field (Invariant)	*Abstract Commands That Change Field*
restore	ABS_Startup, ABS_Init, ABS_SaveState
memory	ABS_Startup, ABS_Init, ABS_save
srk	ABS_Startup, ABS_Init, ABS_TakeOwnership, ABS_OwnerClear, ABS_ForceClear, ABS_RevokeTrust
ek	ABS_Startup, ABS_Init, ABS_CreateEndorsementKeyPair, ABS_CreateRevocableEK, ABS_RevokeTrust
keyGenCtr	ABS_Startup, ABS_Init, ABS_LoadKey2, ABS_CreateWrapKey, ABS_MakeIdentity, ABS_certify
keys	ABS_Startup, ABS_Init, ABS_LoadKey2, ABS_ActivateIdentity, ABS_OwnerClear, ABS_ForceClear, ABS_RevokeTrust
pcrs	ABS_Startup, ABS_Init, ABS_Extend, ABS_sinit, ABS_senter, ABS_PCR_Reset
locality	ABS_Startup, ABS_Init
permFlags	ABS_Startup, ABS_Init, ABS_DisableOwnerClear, ABS_ForceClear, ABS_OwnerClear, ABS_TakeOwnership, ABS_CreateEndorsementKeyPair, ABS_CreateRevocableEK, ABS_RevokeTrust, ABS_PhysicalPresence, ABS_ResetEstablishmentBit
permData	ABS_Startup, ABS_Init, ABS_CreateRevocableEK, ABS_TakeOwnership, ABS_RevokeTrust, ABS_ForceClear, ABS_OwnerClear
stanyFlags	ABS_Startup, ABS_Init
stanyData	ABS_Startup, ABS_Init, ABS_RevokeTrust, ABS_ForceClear, ABS_OwnerClear
stclearFlags	ABS_Startup, ABS_Init, ABS_SetTempDeactivated, ABS_DisableForceClear, ABS_PhysicalPresence
stclearData	ABS_Startup, ABS_Init, ABS_RevokeTrust, ABS_ForceClear, ABS_OwnerClear

Table 5.1. Invariant fields from `tpmAbsState`.

structure. The previous invariant is an example of one such theorem - note the universally quantified variable `c : tpmAbsInput` in the theorem signature.

Invariants on the abstract state are captured in the subtype defined by the `wellFormed?` predicate. Specifically, the definition of instruction execution maps a state of type `(wellFormed?)` to another state of type `(wellFormed?)`. Conditions in the `wellFormed?` predicate include basic structural properties such as the integrity of data for restoring TPM state that will automatically be checked during type checking.

5.3 Additional Theorems

A collection of additional theorems verify detection of replay attacks, spoofed quotes and nonces, and bad signatures. For example, using the following theorem, we can show that a bad nonce, indicating potential replay, is detectable in the quote:

```
bad_nonce: THEOREM
 FORALL (s:(afterStartup?), k:(tpmKey?), n0,n1:(tpmNonce?),
            pm:PCR_SELECTION) :
    quote?(k) AND n0/=n1 =>
       runState(TPM_Quote(k,n0,pm))(s)
       /=
       runState(TPM_Quote(k,n1,pm))(s);
```

Additionally, we confirm that a bad AIK results in a bad quote recognizable in the quote returned by the protocol:

```
bad_signing_key: THEOREM
 FORALL (s:(afterStartup?), k0,k1:(tpmKey?), n:(tpmNonce?),
            pm:PCR_SELECTION) :
    quote?(k0) AND quote?(k1) AND
    private(k0)/=private(k1) =>
       runState(TPM_Quote(k0,n,pm))(s)
       /=
       runState(TPM_Quote(k1,n,pm))(s);
```

Furthermore, we prove that the result of having different PCR values selected is a different quote:

```
bad_pcrs: THEOREM
  FORALL (s:(afterStartup?), k:(tpmKey?), n:(tpmNonce?),
            p0,p1:PCR_SELECTION) :
    quote?(k) AND p0/=p1 =>
       runState(TPM_Quote(k,n,p0))(s)
       /=
       runState(TPM_Quote(k,n,p1))(s);
```

These and similarly formed theorems verify that: (i) bad nonces, AIK signatures and PCR values are detectable; (ii) PCRs record measurement order as well as values; and (iii) `senter` was called to initiate the secure session. These are not properties of individual commands, but of the protocol run's output.

5.4 Protocols

Verification of individual commands is necessary, but it is not sufficient to verify the TPM as a whole. Therefore we must also verify protocols using the TPM. Protocol verification consists of sequencing command execution using the state monad in order to perform complex tasks. While any commands can be sequenced to create a protocol, some sequences make more sense than others. In most cases, the commands sequences necessary for executing TPM protocols are not expressly defined anywhere. Our model provides an opportunity for protocol designers to check their work.

5.4.1 Basic Protocol Verification

Verifying protocols involves using the state monad to sequence command execution to perform more complex tasks. For example, if we create a key, we often follow-up by loading that key into the TPM. Since this includes the use of multiple commands, we can model it as a protocol. Figure 5.3 shows the verification of this protocol.

There are several key parts to verifying protocols. First, the use of bind (`>>=`) and lambda constructs allows one instruction to consume the output of the previous instruction. For example `TPM_LoadKey2` uses the output of `TPM_CreateWrapKey` after it is stored in memory for later use. The use of the `CASE` construct accounts

```
create_load_key_post: THEOREM
 FORALL (state:(afterStartup?),k,p:(tpmKey?),x:nat) :
  LET (a,s) = runState(
              TPM_CreateWrapKey(p,k)
              >>= CPU_saveOutput(x)
              >>= (LAMBDA (a:tpmAbsOutput) :
                   CASES a OF
                     OUT_CreateWrapKey(wk,m) : TPM_LoadKey2(p,wk)
                     ELSE TPM_Noop(a)
                   ENDCASES))
              (state) IN
   createWrapKey?(p,k) AND OUT_CreateWrapKey?(s`memory(x)) AND
   loadKey2?(state,p,wrappedKey(s`memory(x)))
   =>
   a=OUT_LoadKey2(wrappedKey(s`memory(x)),TPM_SUCCESS) AND
   s=state WITH[`keyGenCnt:=state`keyGenCnt+1
                ,`keys:= IF member(key(p),state`keys) OR
                          key(p)=key(TPM_KH_SRK)
                            THEN addKey(wrappedKey(s`memory(x)),state`keys)
                            ELSE state`keys
                          ENDIF
                ,`memory:=s`memory]
```

Figure 5.3. Protocol used to verify key creation and loading.

for the possibility that the previous output is not of the correct type. We are working on mechanisms for eliminating this, thereby cleaning up the protocol representation.

The conditions for proper execution of this sequence of commands involve conditions for proper execution of the commands individually, shown as the predicates `createWrapKey?` and `loadKey2?`. The additional condition in the antecedent, `OUT_CreateWrapKey?(s`memory(x))`, is necessary to verify the `memory` was stored correctly within the `tpmAbsState`. Although this seems obvious based on the use of the `CPU_saveOutput` command, in order to satisfy all type check conditions, this condition must be checked. In the consequent, we ensure that the output

bound to `a` is consistent with the postcondition of `TPM_LoadKey2`, as it is the last command in the sequence. In knowing this is the final output, we know too that the previous commands were correctly executed (and therefore produced non-error outputs). We ensure that the state bound to `s` agrees with updates made to the state from all executed commands. For example, the field `keyGenCnt` is updated by the `TPM_CreateWrapKey` command, `keys` is updated by `TPM_LoadKey2` and `memory` is updated by `CPU_SaveOutput`. However, since `memory` is not an actual TPM field, we ignore the details of what is stored in the updated memory state.

5.4.2 Privacy CA Protocol

The most complex protocol we have verified to date is the Privacy CA Protocol, previously described in Section 2.2. The protocol for generating and certifying an AIK combines all of the previously discussed aspects of our model. Specifically, it uses command predicates as antecedents to verify that a sequence of commands results in the correct quote.

The PVS representation of the protocol from Figure 2.1 that generates the output in Equation 2.1 is shown in Figure 5.4. To verify protocol execution, we first ensure that for all inputs the output bound by the `LET` form to `a` is the quote defined in Equation 2.1 and that the state bound to `s` is the correct state following execution. This tells us the protocol generates the right output.

The `TPM_MakeIdentity` command is called, including parameters of `d` and `k`, a `tpmDigest` and `tpmKey`, respectively. `d` can be any possible `tpmDigest`. `k`, however, is restricted by the `makeIdentity?` predicate to having specific parameters. This key will be the newly created AIK. It may seem strange to pass in an existent key when we are trying to create a new key, but the TPM must specify

```
ca_protocol: THEOREM
 FORALL (state:(afterStartup?), d:(tpmDigest?), k:(tpmKey?),
         n:(tpmNonce?), p:PCR_SELECTION, x,y,z:nat) :
  LET (a,s) = runState(
              TPM_MakeIdentity(d,k)
              >>= CPU_saveOutput(x)
              >>= (LAMBDA (a:tpmAbsOutput) :
                   CASES a OF
                       OUT_MakeIdentity(k,i,m) : CA_certify(k,i)
                      ELSE TPM_Noop(a)
                   ENDCASES)
              >>= CPU_saveOutput(y)
              >>= (LAMBDA (a:tpmAbsOutput) :
                   CASES a OF
                      OUT_Certify(j,d,m) : TPM_ActivateIdentity(j,d)
                      ELSE TPM_Noop(a)
                   ENDCASES)
              >> CPU_read(x)
              >>= (LAMBDA (a:tpmAbsOutput) :
                   CASES a OF
                      OUT_MakeIdentity(k,i,m) : TPM_Quote(k,n,p)
                      ELSE TPM_Noop(a)
                   ENDCASES)
              >>= CPU_saveOutput(z)
              >>= CPU_BuildQuoteFromMem(z,x)) (state) IN
  LET key=idKey(s`memory(x)) IN
   makeIdentity?(state,k) AND
   OUT_MakeIdentity?(s`memory(x)) AND
   certify?(key,idBinding(s`memory(x))) AND
   OUT_Certify?(s`memory(y)) AND
   wellFormedRestore?(s`restore) AND
   activateIdentity?(tpmRestore(s`restore),key,dat(s`memory(y))) AND
   quote?(k) AND OUT_Quote?(s`memory(z))
   =>
   a=OUT_FullQuote(
         tpmQuote(tpmCompositeHash((#select:=p,pcrValue:=s`pcrs#)),
                  n,signed(private(key),clear)),
         tpmIdContents(d,key,signed(private(key),clear)),
         CPU_SUCCESS) AND
   s=state WITH [`keyGenCnt:=2+keyGenCnt(state),
                 `memory:=s`memory]
```

Figure 5.4. Protocol that generates full quote for external appraiser.

the specific parameters of the key that it wants to create. Within the execution of the `TPM_MakeIdentity` command, this key will be modified to have the properties of an AIK. `TPM_MakeIdentity` outputs this AIK, labeled `k` as well as `i:tpmIdContents`, which contains the digest identifying the CA and the public AIK signed with AIK^{-1}.

This output is fed to the `CA_certify` command , which certifies the AIK, called `j`, and returns it along with an EK-encrypted blob containing the digest and newly generated session key, represented as `d`. `TPM_ActivateIdentity` then receives these outputs and attempts to decrypt the session key. In order to generate the quote, `TPM_Quote` uses the AIK (which it gets from the output of `TPM_MakeIdentity`), a `n:tpmNonce`, and a `p:PCR_SELECTION` to output the PCR values signed by the AIK. `CPU_BuildQuoteFromMem` weeds through all of the data that has been generated and presents the quote defined in Equation 2.1.

Due to the number of rewrites required to prove this protocol, this proof takes over 300 hours to run. Given the importance of this protocol, we must prove that it still produces correct output with any major changes to our model. Therefore, the time it takes to prove this protocol is impractical. To make proving this theorem faster, we split the protocol into two parts, the first calling `TPM_MakeIdentity` through `TPM_ActivateIdentity`, and the second starting with `CPU_read` and continuing to the end. This is a convenient place to make the divide since `CPU_read` ignores the output of `TPM_ActivateIdentity`. We use the final state of the first half as the start state for the second, giving us all of the necessary updates stored within the memory.

Given that the preconditions for all commands involved in the CA protocol hold, for any state that the TPM could be in after startup, we prove several things

about the end result. First, a successful structure is output. This means that the structure is a `OUT_FullQuote` type, with a `CPU_SUCCESS` return message. Additionally, the data within this structure is correct: a signed `tpmQuote` – including the correct hash, and nonce values; the signed `tpmIdContents` containing the private portion of the identity key. Finally, we verify that the state has only been modified in ways that we expect. This means that the `keyGenCnt` has been incremented twice, since an AIK was created from the `TPM_MakeIdentity` command, and a symmetric key was created from `CA_certify`. The only other aspect of the state that has changed is `memory`, which does not matter as it is not part of the TPM.

Although the CA protocol is the crowning achievement for our TPM proof collection, we have proved many other theories, named in Table 5.2. This table omits the TCCs as many of them are automatically proved before we even look at them. In cases where there is a second version of a postcondition theorem, `extend_post` and `extend_post2` for example, the second is typically proving that `OUT_Error(TPM_SUCCESS)` is an unreachable case, as previously explained.

PVS Theory	*PVS Theorems*
key	double_inverse, inverse_private, inverse_public, equal_inverse, decrypt_encrypt, decrypt_equal_keys, no_decrypt_unequal_keys, check_sign, load_key, child_if_parent, no_child_if_no_parent
pcr	pcrsResetSelectCorrect, getPCRsCorrectness, antisym
memory	empty_empty, update1, update2
StateMonad	left_identity, right_identity, assocativity
tpm	validatePCRs, check_validatePcrs, gen_quote, gen_cert, init_post, save_state_post, startup_post, startup_after_init, resetMonad, unique_error, set_owner_install_post, owner_set_disable_post, physical_enable_post, physical_disable_post, physical_set_deactivated_post, set_temp_deactivated_post, set_operator_auth_post, take_ownership_post, take_ownership_post2, owner_clear_post, force_clear_post, disable_owner_clear_post, disable_force_clear_post, physical_presence_post, reset_establishment_post, seal_post, unseal_post, unseal_prev_post, unBind_post, unBind_prev_post, create_wrap_key_post, load_key_post, load_key_pred_test, load_key_post2, load_key_post3, create_load_key_post, get_pub_key_post, get_pub_key_post2, create_mig_blob_post, convert_mig_blob_post, authorize_migration_key_post, migrate_key_post, sign_post, sign_pred_test, create_endorsement_key_pair_post, create_revocable_ek_post, revoke_trust_post, revoke_trust_post2, read_pub_ek_post, read_pub_ek_post2, read_pub_ek_take_ownership, owner_read_internal_pub_post, make_identity_post, make_identity_post2, make_cert, activate_identity_post, activate_identity_post2, cert_activate, make_cert_activate_identity, extend_post, extend_post2, antisymmetryMonad, antisymmetryMonad2, antisymmetryMonad3, pcr_read_post, bad_nonce, bad_signing_key, bad_pcrs, check_PCRS, quote_post, quote_with_prev_key, no_senter, pcr_reset_post, pcr_reset_post2,

Table 5.2. Theorems proved.

Chapter 6

Related Works

Our work is based conceptually on the work of Hunt [21, 22], Srivas and Miller [32], and others in the hardware verification community. In a similar manner, we start with a precise engineering description that is textual and graphical in nature. From there, we construct an abstract formal model that we in turn verify rather than verify the details. We assess the validity of our model by executing instructions in both the formal and informal models to determine if we have actually captured requirements.

Most verification work involving the TPM examines systems that use the TPM API [12, 24], not the command set itself. Noteworthy exceptions are works by Delaune et. al. [13, 14], Chen et. al. [8, 9], and Gürgens et. al. [17]. Delaune's work examines properties of functions performed within the TPM using ProVerif for their analysis. While we are attempting to develop an abstract requirements model for the TPM, they focus on verifying cryptographic properties of TPM functions. Their work deals with verifying authentication [14] where they examine a command subset responsible for authentication. Two major differences are their inclusion of session management commands and their decision not to explicitly

model state change. We have chosen to defer session management thus far and explicitly model state change using the state monad described earlier. In their analysis of Microsoft Bitlocker and the envelope protocol [13], they include an attacker while we are looking at functional correctness. These distinctions aside, the abstractions they choose are quite similar to ours even though we are working in higher-order logic in contrast to their use of horn clauses. This is encouraging and suggests that developing a common TPM requirements model may be feasible. It is also worth mentioning here that Ryan's unpublished work [29] is an excellent general introduction to the TPM and its use.

In their first work [9], Chen et. al. analyze the security of the Privacy CA Protocol, considering only cases that use uncorrupted TPMs. Their second work [8] presents an enhanced version of this protocol along with its verification which allows for corrupted TPMs. This work differs from ours in that it focuses exclusively on the CA Protocol and does not handle any other aspect of the TPM. Specifically, there is no model of TPM keys or storage protection. This is especially important when looking at what their model is trying to do in comparison to ours. While we consider only the correctness of the protocol, Chen et. al. perform extensive modeling of the CA Protocol in the presence of an adversary. We look at the protocol *implementation* to ensure that it produces the structure that it should, without considering specific attacks on the structure. We do ensure that bad nonces, PCRs, etc. will be detected, but we do not relate these to security issues. Additionally, this model differs in that there is no automation of proof.

Gürgens and colleagues [17] develop a TPM model using asynchronous product automata (APA) and analyze models using the SH-Verification Tool (SHVT). Their work shares several protocols of interest with ours – secure boot, secure

storage, remote attestation, and data migration – with only remote attestation being described in detail. Like our work they analyze interaction with a Privacy CA, but unlike our work and similar to Delaune, Gürgens includes various kinds of attackers in examining the protocol. Considering multiple attackers with multiple intents is the most interesting contribution of this work. By using a automata model, Gürgens also models state transition explicitly as we do, in contrast with Delaune.

Protocol verification is a discipline unto itself and we claim no substantive contribution there. However, protocols that use the TPM include work by Ramsdell et. al. [28] and Backes [6]. Ramsdell et. al. have done work related to the CAVES protocol using CPSA analysis techniques [28]. The objectives of this work are substantially different, however it is worth acknowledging their complimentary protocol analysis work. Similarly, Backes et. al. [6] verify protocols around the DAA subsystem. Their goals are different, but again this work is complimentary to the verification efforts we are pursuing.

Chapter 7

Conclusions and Future Work

Overall, to verify the functional correctness of the TPM, we have written an abstract specification, defined inputs, outputs and state for a subset of TPM commands, defined and verified requirements for each of these individual commands, and defined and verified several protocols. Modeling these main TPM commands and proving properties of the Privacy CA Protocol has given us confidence that we will be able to effectively model the remainder of the TPM commands and to prove more complex properties of useful protocols.

Although we have modeled the main commands, we have modeled fewer than half of the existing TPM commands. Additionally, the TPM includes the concepts of session management and authorization, which ensure that the TPM commands themselves have permission to run based off of the current state configuration. This entire concept has been excluded from the commands we have implemented. Additionally, the notion of locality within our current model does not contain the full functionality that it does within the TPM.

As our model develops to be closer to that of a TPM implementation, we will be able to model an attacker as well. This is paramount to verifying the TPM.

Currently, most of our proofs are looking at functional correctness. In order to truly verify that the TPM is a trusted platform, we must show that it is impervious to any attacks. As with any attacker model, this will be difficult since the best attacks are those that are unanticipated.

Finally, we will need to complete the concrete half of our bisimulation, as mentioned in Section 2.3. And importantly, we will need to perform the abstraction and concretization functions that form the Galois connection. Without this Galois connection, our abstract model has no credence that it is an authentic model of the TPM.

Although the TCG estimates that TPMs have been embedded in in close to one billion systems [3], few of these systems actually use the capabilities. The release of Windows 8, which requires the presence of a TPM, is expected to drive the use of TPMs to be more mainstream [4, 5]. With the growing attention that the TPM is receiving, having a formal verification of the TPM becomes increasingly important. Consequently, this verification of the TPM and its protocols is imperative. Although the public sector is likely satisfied without such formal methods, this work must be done in order to be able to truly consider the TPM a trusted platform.

References

[1] —. *TCG TPM Specification*. Trusted Computing Group, 3885 SW 153rd Drive, Beaverton, OR 97006, version 1.2 revision 103 edition, July 2007.

[2] —. Attestation Identity Key (AIK) Certificate Enrollment Specification Frequently Asked Questions, September 2011.

[3] —. Trusted Computing Group (TCG) Timeline, October 2012.

[4] —. Windows 8 system requirements, 2013.

[5] W. Ashford. 2012: Will this be the year TPM finally comes of age?, June 2012.

[6] M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 202–215. Ieee, 2008.

[7] D. Challener, K. Yoder, R. Catherman, D. Stafford, and L. V. Doorn. *A Practical Guide to Trusted Computing*. IBM Press, 2007.

[8] L. Chen, M.-F. Lee, and B. Warinschi. Security of the enhanced TCG Privacy-CA solution. In *Trustworthy Global Computing*, pages 121–141. Springer, 2012.

[9] L. Chen and B. Warinschi. Security of the TCG Privacy-CA solution. In *Proceedings of the 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, EUC '10, pages 609–616, Washington, DC, USA, 2010. IEEE Computer Society.

[10] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

[11] G. S. Coker, J. D. Guttman, P. A. Loscocco, J. Sheehy, and B. T. Sniffen. Attestation: Evidence and trust. In *Proceedings of the International Conference on Information and Communications Security*, volume LNCS 5308, 2008.

[12] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 221–236. IEEE, 2009.

[13] S. Delaune, S. Kremer, M. Ryan, and G. Steel. Formal analysis of protocols based on TPM state registers. In *Proceedings of the 24th IEEE Computer Security Foundations Workshop (CSF 2011)*, pages 66–82, 2011.

[14] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel. A formal analysis of authentication in the TPM. In *Proceedings of the Seventh International Workshop on Formal Aspects in Security and Trust (FAST'10)*. Springer, 2010.

[15] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198 – 208, March 1983.

[16] O. Goldreich and Y. Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7:1–32, 1994. 10.1007/BF00195207.

[17] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCGs TPM specification. *Computer Security–ESORICS 2007*, pages 438–453, 2007.

[18] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. In *Proceedings of the Third Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[19] B. Halling and P. Alexander. Verifying A Privacy CA Remote Attestation Protocol. In *Proceedings of NASA Formal Methods (NFM 2013)*, number 7871 in Lecture Notes in Computer Science, pages 398 – 412, Moffett Field, CA, USA, May 2013.

[20] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580,583, 1969.

[21] W. A. Hunt. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[22] W. A. Hunt. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1994.

[23] S. L. Kinney. *Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology)*. Newnes, 2006.

[24] A. H. Lin. *Automated analysis of security APIs*. PhD thesis, Massachusetts Institute of Technology, 2005.

[25] A. Martin. The ten page introduction to Trusted Computing, 2008.

[26] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[27] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proc. of 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer–Verlag.

[28] J. D. Ramsdell, J. D. Guttman, J. K. Millen, and B. O'Hanlon. An analysis of the CAVES Attestation Protocol using CPSA. Technical Remport MTR090213, MITRE, Center for Integrated Intelligence Systems, Bedford, MA, December 2009.

[29] M. Ryan. Introduction to the TPM 1.2. Draft Report, March 2009.

[30] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[31] A. Segall. Introduction to trusted computing, May 2012.

[32] M. K. Srivas and S. P. Miller. Formal verification of the AAMP5 microprocessor. In M. G. Hinchey and J. P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

[33] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

Appendix A

7.1 TPM Theory

```
%% ----
%%
%% TPM Theory
%%
%% Description: Basic model of a TPM using a monadic state model
%%
%% Dependencies:
%%  StateMonad.pvs
%%  ReturnCodes.pvs
%%  memory.pvs
%%  StclearFlags.pvs
%%  startupData.pvs
%%  PermanentData.pvs
%%  StanyData.pvs
%%  StanyFlags.pvs
%%  key.pvs
%%  data.pvs
%%  keyData.pvs
%%  pcr.pvs
%%  PermanentFlags.pvs
%%
%% ----

tpm [ B:TYPE+       % BLOB
    , HV:TYPE+      % Hash value
    , hash:[B->HV]  % Hash function
    ] : THEORY

  BEGIN

  ASSUMING
  % Assume that different blobs always have different hashes
    unique_hash: ASSUMPTION
```

```
    FORALL (b0,b1:B) : hash(b0)=hash(b1) iff b0=b1;
ENDASSUMING

K : TYPE = nat;   % key
RNG : TYPE = int; % random number

IMPORTING ReturnCodes;
IMPORTING startupData[B,K,HV];


%% State monad output type
tpmAbsOutput : DATATYPE
BEGIN
  OUT_Nothing : OUT_Nothing?
  OUT_Error(m:ReturnCode) : OUT_Error?
  OUT_CPUError(m:cpuReturn) : OUT_CPUError?
  OUT_Data_Bind(outData:(tpmBoundData?),m:cpuReturn) : OUT_Data_Bind?
  OUT_Certify(k:(tpmKey?),dat:(tpmAsymCAContents?),m:cpuReturn) : OUT_Certify?
  OUT_FullQuote(quote:(tpmQuote?),idBind:(tpmIdContents?),m:cpuReturn) :
        OUT_FullQuote?
%% Admin Startup and State (3)
  OUT_Init(m:ReturnCode) : OUT_Init?
  OUT_SaveState(m:ReturnCode) : OUT_SaveState?
  OUT_Startup(m:ReturnCode) : OUT_Startup?
%% Admin Opt-in (5)
  OUT_SetOwnerInstall(m:ReturnCode) : OUT_SetOwnerInstall?
  OUT_OwnerSetDisable(m:ReturnCode) : OUT_OwnerSetDisable?
  OUT_PhysicalEnable(m:ReturnCode) : OUT_PhysicalEnable?
  OUT_PhysicalDisable(m:ReturnCode) : OUT_PhysicalDisable?
  OUT_PhysicalSetDeactivated(m:ReturnCode) : OUT_PhysicalSetDeactivated?
  OUT_SetTempDeactivated(m:ReturnCode) : OUT_SetTempDeactivated?
  OUT_SetOperatorAuth(m:ReturnCode) : OUT_SetOperatorAuth?
%% Admin Ownership (6)
  OUT_TakeOwnership(srk:(tpmKey?),m:ReturnCode) : OUT_TakeOwnership?
  OUT_OwnerClear(m:ReturnCode) : OUT_OwnerClear?
  OUT_ForceClear(m:ReturnCode) : OUT_ForceClear?
  OUT_DisableOwnerClear(m:ReturnCode) : OUT_DisableOwnerClear?
  OUT_DisableForceClear(m:ReturnCode) : OUT_DisableForceClear?
  OUT_PhysicalPresence(m:ReturnCode) : OUT_PhysicalPresence?
  OUT_ResetEstablishmentBit(m:ReturnCode) : OUT_ResetEstablishmentBit?
%% Storage Functions (10)
  OUT_Seal(sealedData:(tpmStoredData?),m:ReturnCode) : OUT_Seal?
  OUT_Unseal(secret:tpmData,m:ReturnCode) : OUT_Unseal?
  OUT_UnBind(outData:tpmData,m:ReturnCode) : OUT_UnBind?
  OUT_CreateWrapKey(wrappedKey:(tpmKey?),m:ReturnCode) :
        OUT_CreateWrapKey?
  OUT_LoadKey2(inkeyHandle:(tpmKey?),m:ReturnCode) : OUT_LoadKey2?
  OUT_GetPubKey(pubKey:(tpmPubkey?),m:ReturnCode) : OUT_GetPubKey?
%% Migration (11)
  OUT_CreateMigrationBlob(random:int,outData:tpmData,m:ReturnCode) :
```

```
        OUT_CreateMigrationBlob?
  OUT_ConvertMigrationBlob(outData:(tpmStoreAsymkey?),m:ReturnCode) :
        OUT_ConvertMigrationBlob?
  OUT_AuthorizeMigrationKey(outData:(tpmMigKeyAuth?),m:ReturnCode) :
        OUT_AuthorizeMigrationKey?
  OUT_MigrateKey(outData:tpmData,m:ReturnCode) : OUT_MigrateKey?
%% Cryptographic Functions (13)
  OUT_Sign(sig:tpmData,m:ReturnCode) : OUT_Sign?
%% Endorsement Key Handling (14)
  OUT_CreateEndorsementKeyPair(pubEk:(tpmKey?),checksum:(tpmDigest?),
        m:ReturnCode) : OUT_CreateEndorsementKeyPair?
  OUT_CreateRevocableEK(pubEK:(tpmPubkey?),checksum:(tpmDigest?),
        outputEKreset:(tpmNonce?),m:ReturnCode) : OUT_CreateRevocableEK?
  OUT_RevokeTrust(m:ReturnCode) : OUT_RevokeTrust?
  OUT_ReadPubek(pubEk:(tpmKey?),checksum:(tpmDigest?),m:ReturnCode) :
        OUT_ReadPubek?
  OUT_OwnerReadInternalPub(k:(tpmPubkey?),m:ReturnCode) :
        OUT_OwnerReadInternalPub?
%% Identity Creation and Activation (15)
  OUT_MakeIdentity(idKey:(tpmKey?),idBinding:(tpmIdContents?),m:ReturnCode) :
        OUT_MakeIdentity?
 OUT_ActivateIdentity(symmKey:(tpmSessKey?),m:ReturnCode) : OUT_ActivateIdentity?
%% Integrity Collection and Reporting (16)
  OUT_Extend(outDigest:PCR,m:ReturnCode) : OUT_Extend?
  OUT_PCRRead(outDigest:PCR,m:ReturnCode) : OUT_PCRRead?
  OUT_Quote(pcrData:list[PCR],sig:(tpmQuote?),m:ReturnCode) : OUT_Quote?
  OUT_PCR_Reset(m:ReturnCode) : OUT_PCR_Reset?
END tpmAbsOutput;

IMPORTING memory[tpmAbsOutput,OUT_Nothing];

%% State monad input type.  All inputs are in the form of a command with
%% parameters.
tpmAbsInput : DATATYPE
BEGIN
%% Admin Startup and State commands (3)
  ABS_Init : ABS_Init?
  ABS_SaveState : ABS_SaveState?
  ABS_Startup(startupType : TPM_STARTUP_TYPE) : ABS_Startup?
%% Admin Opt-in (5)
  ABS_SetOwnerInstall(state:bool) : ABS_SetOwnerInstall?
  ABS_OwnerSetDisable(disableState:bool) : ABS_OwnerSetDisable?
  ABS_PhysicalEnable : ABS_PhysicalEnable?
  ABS_PhysicalDisable : ABS_PhysicalDisable?
  ABS_PhysicalSetDeactivated(state:bool) : ABS_PhysicalSetDeactivated?
  ABS_SetTempDeactivated : ABS_SetTempDeactivated?
  ABS_SetOperatorAuth(opAuth:(tpmSecret?)) : ABS_SetOperatorAuth?
%% Admin Ownership Commands (6)
  ABS_TakeOwnership(srk:(tpmKey?)) : ABS_TakeOwnership?
  ABS_OwnerClear : ABS_OwnerClear?
```

```
  ABS_ForceClear : ABS_ForceClear?
  ABS_DisableOwnerClear : ABS_DisableOwnerClear?
  ABS_DisableForceClear : ABS_DisableForceClear?
  ABS_PhysicalPresence(p:PHYSPRES) : ABS_PhysicalPresence?
  ABS_ResetEstablishmentBit : ABS_ResetEstablishmentBit?
%% Protected Storage Commands (10)
  ABS_Seal(k:(tpmKey?),pcrInfo:(tpmPCRInfoLong?),inData:tpmData) : ABS_Seal?
  ABS_Unseal(parent:(tpmKey?),inData:(tpmStoredData?)) : ABS_Unseal?
  ABS_UnBind(key:(tpmKey?),inData:(tpmBoundData?)) : ABS_UnBind?
  ABS_CreateWrapKey(parentH,keyInfo:(tpmKey?)) : ABS_CreateWrapKey?
  ABS_LoadKey2(parent,inKey:(tpmKey?)): ABS_LoadKey2?
  ABS_GetPubKey(key:(tpmKey?)): ABS_GetPubKey?
%% Migration Commands (11)
  ABS_CreateMigrationBlob(p:(tpmKey?),m:migrateScheme,migKey:(tpmMigKeyAuth?),
        encData:(tpmKey?)) : ABS_CreateMigrationBlob?
  ABS_ConvertMigrationBlob(parent:(tpmKey?),inData:(tpmMigrateAsymkey?),
        random:int) : ABS_ConvertMigrationBlob?
  ABS_AuthorizeMigrationKey(migKey:(tpmKey?),migScheme:(tpmMigScheme?)) :
        ABS_AuthorizeMigrationKey?
  ABS_MigrateKey(ma,pub:(tpmKey?),inData:tpmData) : ABS_MigrateKey?
%% Cryptographic Commands (13)
  ABS_Sign(keyHandle:(tpmKey?),areaToSign:tpmData) : ABS_Sign?
%% Endorsement Key Handling (14)
  ABS_CreateEndorsementKeyPair(antiReplay:(tpmNonce?),keyInfo:(tpmKey?)) :
        ABS_CreateEndorsementKeyPair?
  ABS_CreateRevocableEK(antiReplay:(tpmNonce?),keyInfo:(tpmKey?),
        genReset:bool,inputEKreset:(tpmNonce?)) : ABS_CreateRevocableEK?
  ABS_RevokeTrust(EKReset:(tpmNonce?)) : ABS_RevokeTrust?
  ABS_ReadPubek(n:(tpmNonce?)) : ABS_ReadPubek?
  ABS_OwnerReadInternalPub(k:(tpmKey?)) : ABS_OwnerReadInternalPub?
%% Identity Creation and Activation (15)
  ABS_MakeIdentity(CADigest:(tpmDigest?),idKey:(tpmKey?)) : ABS_MakeIdentity?
  ABS_ActivateIdentity(aik:(tpmKey?),b:(activateIdentityBlob?)) : ABS_ActivateIdentity?
%% Integrity Collection and Reporting (16)
  ABS_Extend(pcrNum:PCRINDEX,d:HV) : ABS_Extend?
  ABS_PCRRead(ind:PCRINDEX) : ABS_PCRRead?
  ABS_Quote(aik:(tpmKey?),nonce:(tpmNonce?),pm:PCR_SELECTION) : ABS_Quote?
  ABS_PCR_Reset(pcrSelect:PCR_SELECTION) : ABS_PCR_Reset?
%% Software Commands
  ABS_senter : ABS_senter? % implemented all actions as one senter
  ABS_sinit : ABS_sinit? % partially implemented
  ABS_save(i:nat,v:tpmAbsOutput) : ABS_save?
  ABS_read(i:nat) : ABS_read?
  ABS_Data_Bind(k:(tpmKey?),d:tpmData) : ABS_Data_Bind?
%% CA Commands
 ABS_certify(aik:(tpmKey?),certReq:(tpmIdContents?)) : ABS_certify?
%% Invented, imaginary Commands
  noopCom : noopCom?
END tpmAbsInput;
```

```
sinit : B;    %% sinit blob instance for measurement
mle   : B;    %% mle blob instance for measurement
rand  : RNG;  %% random number

%% Initial key values not generated by TPM
ekKeyVal   : K = 1;
srkKeyVal  : K = 2;
caKeyVal   : K = 3;
initKeyVal : K = 100;  %% Initial key count value for initializing TPM

%% Key definitions that make ek and srk values asymmetric keys.
ekVal:(tpmKey?) = tpmKey(ekKeyVal,storage,keyFlagsF,always,keyParmsDef,
                         pcrInfoLongDefault,1,storeAsymkeyDefault,clear);
srkVal:(tpmKey?) = tpmKey(srkKeyVal,storage,keyFlagsF,always,keyParmsDef,
                         pcrInfoLongDefault,1,storeAsymkeyDefault,clear);
caVal:(tpmKey?) = tpmKey(caKeyVal,storage,keyFlagsF,always,keyParmsDef,
                         pcrInfoLongDefault,1,storeAsymkeyDefault,clear);
                         % Certificate Authority key

%% Abstract TPM State with keys, PCR array and locality
tpmAbsState : TYPE = [#
                      restore : restoreStateData
                    , memory : mem
                    , srk : (tpmKey?)
                    , ek : (tpmKey?)
                    , keyGenCnt : K
                    , keys : KEYSET
                    , pcrs : PCRVALUES
                    , locality : LOCALITY
                    , permFlags : PermFlags
                    , permData : PermData
                    , stanyFlags : StanyFlags
                    , stanyData : StanyData
                    , stclearFlags : StclearFlags
                    , stclearData : StclearData
                    #];

%% Well formedness condition for abstract states.  Currently unused, but we
%% should show that forall commands, well formed input generates well formed
%% output.
wellFormed?(s:tpmAbsState):bool = wellFormedRestore?(restore(s));

IMPORTING StateMonad[tpmAbsOutput,tpmAbsState];

%% Define some common TPM states and state operations

%% Unknown state
tpmUnknown : tpmAbsState

%% Power on state after init is raised by hardware.
```

```
tpmPostInit : (wellFormed?) = (#
                    , pcrs:=pcrsPower
                    , locality:=4
                    , keys:=emptyset
                    , srk:=privateKey(srkVal)
                    , ek:=privateKey(ekVal)
                    , keyGenCnt:=initKeyVal
                    , memory:=empty
                    , restore:=initSaveData
                    , permFlags:=permFlagsDefault
                    , permData:=permDataInit
                    , stanyFlags:=stanyFlagsInit WITH [‘postInitialize:=TRUE]
                    , stanyData:=stanyDataInit
                    , stclearFlags:=stclearFlagsInit
                    , stclearData :=stclearDataInit
                    #);

%% Standard initial state following startup command with the TPM_ST_CLEAR
%% option set.  Note that this should be checked against the spec before
%% asserting goodness.
tpmStartup : (wellFormed?) = (#
                      pcrs:=pcrsReset(allResetAccess)
                    , locality:=4
                    , keys:=emptyset
                    , srk:=privateKey(srkVal)
                    , ek:=privateKey(ekVal)
                    , keyGenCnt:=initKeyVal
                    , memory:=empty
                    , restore:=initSaveData
                    , permFlags:=permFlagsDefault
                    , permData:=permDataInit
                    , stanyFlags:=stanyFlagsInit WITH [‘postInitialize:=FALSE]
                    , stanyData:=stanyDataInit
                    , stclearFlags:=stclearFlagsInit
                    , stclearData :=stclearDataInit
                    #);

%% Generate a new state from restore data.  Basically this is a clear
%% restart with pcrs, keys, and pcr flags coming from the restore
%% data.  Note that this function assumes valid data and will behave
%% badly otherwise
tpmRestore(rd:(wellFormedRestore?)) : (wellFormed?) = (#
                      pcrs:=pcrs(rd)
                    , locality:=4
                    , keys:=keys(rd)
                    , srk:=privateKey(srkVal)
                    , ek:=privateKey(ekVal)
                    , keyGenCnt:=initKeyVal
                    , memory:=empty
                    , restore:=rd
```

```
                    , permFlags:=permFlags(rd)
                    , permData:=permData(rd)
                    , stanyFlags:=stanyFlags(rd)
                    , stanyData:=stanyData(rd)
                    , stclearFlags:=stclearFlagsInit
                    , stclearData :=stclearDataInit
                    #)

%% Predicate to determine if startup has occured.  Used as types
%% (afterInit?) is the set of states occuring immediately after TPM_Init
%% (afterStartup?) is the set of states occuring immediately after
%% TPM_Startup
afterInit?(s:tpmAbsState):bool = postInitialize(stanyFlags(s));
afterStartup?(s:tpmAbsState):bool = NOT postInitialize(stanyFlags(s));

%% Standard operations on TPM state definition above.  All such
%% functions end with State to indicate they operate on the state
%% value rather than on the TPM monad.  Note that key set and pcr
%% manipulation functions are defined externally in key.pvs and
%% pcr.pvs respectively

%% Reset PCRs as performed by SENTER.
pcrsResetSenterState(s:tpmAbsState) : tpmAbsState =
  s WITH [`pcrs := pcrsSenter(pcrs(s),pcrAttrib(permData(s)))];

%% Decrease locality value
changeLocalityState(s:tpmAbsState) : tpmAbsState =
  s WITH [`locality := IF locality(s) > 0
                       THEN locality(s) - 1
                       ELSE 0
                       ENDIF];

%% Generate a new key
genKeyState(s:tpmAbsState) : tpmAbsState =
  s WITH [`keyGenCnt := keyGenCnt(s)+1];

saveState(s:tpmAbsState) : tpmAbsState =
  s WITH [`restore:=saveState(keys(s),ek(s),srk(s),keyGenCnt(s),pcrs(s),
                              permFlags(s),permData(s),stanyFlags(s),stanyData(s),
                              stclearFlags(s),stclearData(s))];

setOwnerInstallState(s:tpmAbsState,state:bool) : tpmAbsState =
  IF s`permFlags`ownership
  THEN s
  ELSIF s`stclearFlags`physicalPresence
  THEN s WITH [`permFlags`ownership:=state]
  ELSE s
  ENDIF;

setOwnerInstallOut(s:tpmAbsState,state:bool) : tpmAbsOutput =
```

```
  IF s‘permFlags‘ownership
  THEN OUT_SetOwnerInstall(TPM_SUCCESS)
  ELSIF s‘stclearFlags‘physicalPresence
  THEN OUT_SetOwnerInstall(TPM_SUCCESS)
  ELSE OUT_Error(TPM_BAD_PRESENCE)
  ENDIF;

ownerSetDisableState(s:tpmAbsState,disableState:bool) : tpmAbsState =
  s WITH [‘permFlags‘disable:=disableState]

ownerSetDisableOut(s:tpmAbsState,disableState:bool) : tpmAbsOutput =
  OUT_OwnerSetDisable(TPM_SUCCESS);

physicalEnableState(s:tpmAbsState) : tpmAbsState =
  IF not s‘stclearFlags‘physicalPresence
  THEN s
  ELSE s WITH [‘permFlags‘disable:=FALSE]
  ENDIF;

physicalEnableOut(s:tpmAbsState) : tpmAbsOutput =
  IF not s‘stclearFlags‘physicalPresence
  THEN OUT_Error(TPM_BAD_PRESENCE)
  ELSE OUT_PhysicalEnable(TPM_SUCCESS)
  ENDIF;

physicalDisableState(s:tpmAbsState) : tpmAbsState =
  IF not s‘stclearFlags‘physicalPresence
  THEN s
  ELSE s WITH [‘permFlags‘disable:=TRUE]
  ENDIF;

physicalDisableOut(s:tpmAbsState) : tpmAbsOutput =
  IF not s‘stclearFlags‘physicalPresence
  THEN OUT_Error(TPM_BAD_PRESENCE)
  ELSE OUT_PhysicalDisable(TPM_SUCCESS)
  ENDIF;

physicalSetDeactivatedState(s:tpmAbsState,state:bool) : tpmAbsState =
  IF not s‘stclearFlags‘physicalPresence
  THEN s
  ELSE s WITH [‘permFlags‘disable:=state]
  ENDIF;

physicalSetDeactivatedOut(s:tpmAbsState,state:bool) : tpmAbsOutput =
  IF not s‘stclearFlags‘physicalPresence
  THEN OUT_Error(TPM_BAD_PRESENCE)
  ELSE OUT_PhysicalSetDeactivated(TPM_SUCCESS)
  ENDIF;

setTempDeactivatedState(s:tpmAbsState) : tpmAbsState =
```

```
  IF not s‘permFlags‘operator
  THEN s
  ELSIF not s‘stclearFlags‘physicalPresence
  THEN s
  ELSE s WITH [‘stclearFlags‘deactivated:=TRUE]
  ENDIF;

setTempDeactivatedOut(s:tpmAbsState) : tpmAbsOutput =
  IF not s‘permFlags‘operator
  THEN OUT_Error(TPM_NOOPERATOR)
  ELSIF not s‘stclearFlags‘physicalPresence
  THEN OUT_Error(TPM_BAD_PRESENCE)
  ELSE OUT_SetTempDeactivated(TPM_SUCCESS)
  ENDIF;

setOperatorAuthState(s:tpmAbsState,opAuth:(tpmSecret?)) : tpmAbsState =
  IF not s‘stclearFlags‘physicalPresence
  THEN s
  ELSE s WITH [‘permData‘operatorAuth:=opAuth
       ,‘permFlags‘operator:=TRUE]
  ENDIF;

setOperatorAuthOut(s:tpmAbsState,opAuth:(tpmSecret?)) : tpmAbsOutput =
  IF not s‘stclearFlags‘physicalPresence
  THEN OUT_Error(TPM_BAD_PRESENCE)
  ELSE OUT_SetOperatorAuth(TPM_SUCCESS)
  ENDIF;


takeOwnership?(s:tpmAbsState,k:(tpmKey?)) : bool =
  COND
    i(s‘permData‘ownerAuth)/=INVALIDAUTH -> FALSE,
    not s‘permFlags‘ownership -> FALSE,
    not goodkey?(key(s‘ek)) -> FALSE,
    not storage?(keyUsage(k)) -> FALSE,
    migratable(keyFlags(k)) -> FALSE,
    not RSA?(algoId(algoParms(k))) -> FALSE,
    s‘permFlags‘FIPS AND never?(authDataUsage(k))-> FALSE,
    ELSE -> TRUE
  ENDCOND;

takeOwnershipState(s:tpmAbsState,k:(tpmKey?)) : tpmAbsState =
  COND
    NOT takeOwnership?(s,k) -> s,
    ELSE -> LET A2=tempAuthData IN
            LET asymkey=tpmStoreAsymkey(A2,
                                migrationAuth(encDat(k)),
                                pubDataDigest(encDat(k)),
                                privKey(encDat(k)),
                                crs(encDat(k))) IN
```

```
            LET K1=tpmKey(key(k),keyUsage(k),keyFlags(k),
                      authDataUsage(k),algoParms(k),
                      PCRInfo(k),wrappingKey(k),asymkey,
                      crs(k)) IN
            s WITH [`srk:=K1
                   ,`permFlags(readPubek):=FALSE]
  ENDCOND;

takeOwnershipOut(s:tpmAbsState,k:(tpmKey?)) : tpmAbsOutput =
  IF NOT takeOwnership?(s,k)
  THEN IF i(s`permData`ownerAuth)/=INVALIDAUTH
       THEN OUT_Error(TPM_OWNER_SET)
     ELSIF not s`permFlags`ownership
       THEN OUT_Error(TPM_INSTALL_DISABLED)
     ELSIF not goodkey?(key(s`ek))
       THEN OUT_Error(TPM_NO_ENDORSEMENT)
     ELSIF not storage?(keyUsage(k))
       THEN OUT_Error(TPM_INVALID_KEYUSAGE)
     ELSIF migratable(keyFlags(k))
       THEN OUT_Error(TPM_INVALID_KEYUSAGE)
     ELSIF not RSA?(algoId(algoParms(k)))
       THEN OUT_Error(TPM_BAD_KEY_PROPERTY)
     ELSIF s`permFlags`FIPS AND never?(authDataUsage(k))
       THEN OUT_Error(TPM_NOTFIPS)
     ELSE OUT_Error(TPM_SUCCESS) % Should be unreachable case.
     ENDIF
  ELSE LET A2=tempAuthData IN
       LET asymkey=tpmStoreAsymkey(A2,
                     migrationAuth(encDat(k)),
                     pubDataDigest(encDat(k)),
                     privKey(encDat(k)),
                     crs(encDat(k))) IN
       LET K1=tpmKey(key(k),keyUsage(k),keyFlags(k),authDataUsage(k),
                    algoParms(k),PCRInfo(k),wrappingKey(k),asymkey,
                    crs(k)) IN
       OUT_TakeOwnership(K1,TPM_SUCCESS)
  ENDIF;


clear(s:tpmAbsState) : tpmAbsState = % Not fully implemented.
  s WITH [`keys:=emptyset
         ,`permData(ownerAuth):=tpmSecret(INVALIDAUTH)
         ,`srk:=tpmKey(0,keyUsage(s`srk),keyFlags(s`srk),authDataUsage(s`srk),
                      algoParms(s`srk),PCRInfo(s`srk),wrappingKey(s`srk),
                      encDat(s`srk),crs(s`srk))
         ,`permData(tpmProof):=tpmSecret(INVALIDPROOF)
         ,`permData(operatorAuth):=tpmSecret(INVALIDAUTH)
         ,`stanyData:=stanyDataInit
         ,`stclearData:=stclearDataInit WITH [`PCR:=PCR(stclearData(s))]
         ,`permFlags(disable):=disableDef
```

```
         ,‘permFlags(deactivated):=deactivatedDef
         ,‘permFlags(readPubek):=readPubekDef
         ,‘permFlags(disableOwnerClear):=disableOwnerClearDef
         ,‘permFlags(disableFullDALogicInfo):=disableFullDALogicInfoDef
         ,‘permFlags(allowMaintenance):=allowMaintenanceDef
         ,‘permFlags(readSRKPub):=readSRKPubDef
         ,‘permFlags(ownership):=TRUE
         ,‘permFlags(operator):=FALSE
         ,‘permFlags(maintenanceDone):=FALSE
         ]

ownerClearState(s:tpmAbsState) : tpmAbsState =
  COND
    s‘permFlags‘disableOwnerClear -> s,
    ELSE -> clear(s)
  ENDCOND;

ownerClearOut(s:tpmAbsState) : tpmAbsOutput =
  COND
    disableOwnerClear(permFlags(s)) -> OUT_Error(TPM_CLEAR_DISABLED),
    ELSE -> OUT_OwnerClear(TPM_SUCCESS)
  ENDCOND;

forceClearState(s:tpmAbsState) : tpmAbsState =
  IF s‘stclearFlags‘physicalPresence
  THEN IF s‘stclearFlags‘disableForceClear
       THEN s
       ELSE clear(s)
       ENDIF
  ELSE s
  ENDIF

forceClearOut(s:tpmAbsState) : tpmAbsOutput =
  IF s‘stclearFlags‘physicalPresence
  THEN IF s‘stclearFlags‘disableForceClear
       THEN OUT_Error(TPM_CLEAR_DISABLED)
       ELSE OUT_ForceClear(TPM_SUCCESS)
       ENDIF
  ELSE OUT_Error(TPM_BAD_PRESENCE)
  ENDIF

disableOwnerClearState(s:tpmAbsState) : tpmAbsState =
  s WITH [‘permFlags(disableOwnerClear):=TRUE];

disableOwnerClearOut(s:tpmAbsState) : tpmAbsOutput =
  OUT_DisableOwnerClear(TPM_SUCCESS)

disableForceClearState(s:tpmAbsState) : tpmAbsState =
  s WITH [‘stclearFlags(disableForceClear):=TRUE];
```

```
disableForceClearOut(s:tpmAbsState) : tpmAbsOutput =
  OUT_DisableForceClear(TPM_SUCCESS)

physicalPresenceState(s:tpmAbsState,p:PHYSPRES) : tpmAbsState =
  IF member(LIFETIME_LOCK,p) OR member(HW_ENABLE,p) OR member(CMD_ENABLE,p)
     OR member(HW_DISABLE,p) OR member(CMD_DISABLE,p)
  THEN IF s`permFlags`physicalPresenceLifetimeLock
       THEN s
       ELSIF member(LOCK,p) OR member(PRESENT,p) OR member(NOTPRESENT,p)
       THEN s
       ELSIF member(HW_ENABLE,p) AND member(HW_DISABLE,p)
       THEN s
       ELSIF member(CMD_ENABLE,p) AND member(CMD_DISABLE,p)
       THEN s
       ELSE s WITH [`permFlags`physicalPresenceHWEnable:=
                       IF member(HW_ENABLE,p) THEN TRUE
                       ELSIF member(HW_DISABLE,p) THEN FALSE
                       ELSE s`permFlags`physicalPresenceHWEnable ENDIF,
                    `permFlags`physicalPresenceCMDEnable:=
                       IF member(CMD_ENABLE,p) THEN TRUE
                       ELSIF member(HW_DISABLE,p) THEN FALSE
                       ELSE s`permFlags`physicalPresenceCMDEnable ENDIF,
                    `permFlags`physicalPresenceLifetimeLock:=
                       IF member(LIFETIME_LOCK,p) THEN TRUE
                       ELSE s`permFlags`physicalPresenceLifetimeLock ENDIF]
       ENDIF
  ELSIF member(LOCK,p) OR member(PRESENT,p) OR member(NOTPRESENT,p)
  THEN IF member(LIFETIME_LOCK,p) OR member(HW_ENABLE,p) OR
          member(CMD_ENABLE,p)OR member(HW_DISABLE,p) OR
          member(CMD_DISABLE,p)
       THEN s
       ELSIF s`permFlags`physicalPresenceCMDEnable=FALSE
       THEN s
       ELSIF member(LOCK,p) AND member(PRESENT,p)
       THEN s
       ELSIF member(PRESENT,p) AND member(NOTPRESENT,p)
       THEN s
       ELSIF s`stclearFlags`physicalPresenceLock
       THEN s
       ELSIF member(LOCK,p)
       THEN s WITH [`stclearFlags`physicalPresence:=FALSE
                   ,`stclearFlags`physicalPresenceLock:=TRUE]
       ELSIF member(PRESENT,p)
       THEN s WITH [`stclearFlags`physicalPresence:=TRUE]
       ELSIF member(NOTPRESENT,p)
       THEN s WITH [`stclearFlags`physicalPresence:=FALSE]
       ELSE s % Should be unreachable case.
       ENDIF
  ELSE s
  ENDIF;
```

```
physicalPresenceOut(s:tpmAbsState,p:PHYSPRES) : tpmAbsOutput =
  IF member(LIFETIME_LOCK,p) OR member(HW_ENABLE,p) OR member(CMD_ENABLE,p)
     OR member(HW_DISABLE,p) OR member(CMD_DISABLE,p)
  THEN IF s`permFlags`physicalPresenceLifetimeLock
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(LOCK,p) OR member(PRESENT,p) OR member(NOTPRESENT,p)
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(HW_ENABLE,p) AND member(HW_DISABLE,p)
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(CMD_ENABLE,p) AND member(CMD_DISABLE,p)
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSE OUT_PhysicalPresence(TPM_SUCCESS)
       ENDIF
  ELSIF member(LOCK,p) OR member(PRESENT,p) OR member(NOTPRESENT,p)
  THEN IF member(LIFETIME_LOCK,p) OR member(HW_ENABLE,p) OR
          member(CMD_ENABLE,p)OR member(HW_DISABLE,p) OR
          member(CMD_DISABLE,p)
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF s`permFlags`physicalPresenceCMDEnable=FALSE
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(LOCK,p) AND member(PRESENT,p)
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(PRESENT,p) AND member(NOTPRESENT,p)
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF s`stclearFlags`physicalPresenceLock
       THEN OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(LOCK,p)
       THEN OUT_PhysicalPresence(TPM_SUCCESS)
       ELSIF member(PRESENT,p)
       THEN OUT_PhysicalPresence(TPM_SUCCESS)
       ELSIF member(NOTPRESENT,p)
       THEN OUT_PhysicalPresence(TPM_SUCCESS)
       ELSE OUT_Error(TPM_BAD_PARAMETER) % Should be unreachable case.
       ENDIF
  ELSE OUT_Error(TPM_BAD_PARAMETER)
  ENDIF

resetEstablishment?(s:tpmAbsState) : bool =
  s`locality=3 or s`locality=4;

resetEstablishmentBitState(s:tpmAbsState) : tpmAbsState =
  IF resetEstablishment?(s)
  THEN s WITH [`permFlags(tpmEstablished):=FALSE]
  ELSE s
  ENDIF

resetEstablishmentBitOut(s:tpmAbsState) : tpmAbsOutput =
  IF resetEstablishment?(s)
  THEN OUT_ResetEstablishmentBit(TPM_SUCCESS)
```

```
  ELSE OUT_Error(TPM_BAD_LOCALITY)
  ENDIF

seal?(k:(tpmKey?)) : bool =
  storage?(keyUsage(k)) AND not(migratable(keyFlags(k)))

sealOut(s:tpmAbsState,kH:(tpmKey?),p:(tpmPCRInfoLong?),inData:tpmData) :
 tpmAbsOutput =
  IF not seal?(kH)
  THEN OUT_Error(TPM_INVALID_KEYUSAGE)
  ELSE LET h2=tpmCompositeHash((#select:=creationPCRSelect(p),
                                pcrValue:=s`pcrs#)) IN
       LET s1=tpmStoredData(tpmPCRInfoLong(
                                 s`stanyFlags`localityModifier,
                                 locAtRelease(p),
                                 creationPCRSelect(p),
                                 releasePCRSelect(p),
                                 h2,
                                 digAtRelease(p)),
                            tpmSealedNull,
                            clear) IN
       LET a1=tempAuthData,
           h3=tpmDigest(cons(s1,null),clear) IN
       LET S3=tpmSealedData(a1,s`permData`tpmProof,h3,inData,
                            encrypted(key(kH),clear)) IN
       LET S1=tpmStoredData(sealInfo(s1),S3,clear) IN
       OUT_Seal(S1,TPM_SUCCESS)
  ENDIF

unseal?(p:(tpmKey?)) : bool =
  storage?(keyUsage(p)) AND not(migratable(keyFlags(p)))

% Output secret unsealed with an asymmetric key and PCRs
unsealOut(s:tpmAbsState,parent:(tpmKey?),inData:(tpmStoredData?)):tpmAbsOutput =
  LET d=decrypt(key(parent),inData) IN
  LET d1:(tpmSealedData?)=encData(d),
      S2=tpmStoredData(sealInfo(inData),tpmSealedNull,clear),
      h2=tpmCompositeHash((#select:=releasePCRSelect(sealInfo(inData)),
                           pcrValue:=s`pcrs#)) IN
  IF not unseal?(parent)
  THEN OUT_Error(TPM_INVALID_KEYUSAGE)
  ELSIF not crs(d)=clear
  THEN OUT_CPUError(CPU_DECRYPT_ERROR)
  ELSIF not(tpmProof(d1)=tpmProof(s`permData) AND
            tpmDigest(cons(S2,null),clear)=storedDigest(d1))
  THEN OUT_Error(TPM_NOTSEALED_BLOB)
  ELSIF not locAtRelease(sealInfo(S2))=localityModifier(s`stanyFlags)
  THEN OUT_Error(TPM_BAD_LOCALITY)
  ELSIF not h2=digAtRelease(sealInfo(S2))
  THEN OUT_Error(TPM_WRONGPCRVAL)
```

```
  ELSE OUT_Unseal(data(d1),TPM_SUCCESS)
  ENDIF;

unBind?(k:(tpmKey?)) : bool =
  legacy?(keyUsage(k)) OR bind?(keyUsage(k))

% Output decrypted key
unBindOut(s:tpmAbsState,k:(tpmKey?),d:(tpmBoundData?)):tpmAbsOutput=
  LET d1:(tpmBoundData?)=decrypt(private(k),d) IN
  IF not unBind?(k)
  THEN OUT_Error(TPM_INVALID_KEYUSAGE)
  ELSIF not clear?(crs(d1))
  THEN OUT_CPUError(CPU_DECRYPT_ERROR)
  ELSE OUT_UnBind(payloadData(d1),TPM_SUCCESS)
  ENDIF;

createWrapKey?(p,k:(tpmKey?)): bool =
  storage?(keyUsage(p)) AND
  IF migratable(keyFlags(p)) AND not(migratable(keyFlags(k)))
  THEN FALSE
  ELSE IF identity?(keyUsage(k)) OR authChange?(keyUsage(k))
       THEN FALSE
       ELSE not(migrateAuthority(keyFlags(k)))
       ENDIF
  ENDIF;

createWrapKeyState(s:tpmAbsState,p,k:(tpmKey?)) : tpmAbsState =
  IF not createWrapKey?(p,k)
  THEN s
  ELSIF s`permFlags`FIPS AND (never?(authDataUsage(k))
                              OR legacy?(keyUsage(k)))
  THEN s
  ELSIF (storage?(keyUsage(k)) or migrate?(keyUsage(k)))
        AND not RSA?(algoId(algoParms(k)))
  THEN s
  ELSE genKeyState(s)
  ENDIF;

createWrapKeyOut(s:tpmAbsState,p,k:(tpmKey?)) : tpmAbsOutput =
  IF not createWrapKey?(p,k)
  THEN OUT_Error(TPM_INVALID_KEYUSAGE)
  ELSIF s`permFlags`FIPS AND (never?(authDataUsage(k))
                              OR legacy?(keyUsage(k)))
  THEN OUT_Error(TPM_NOTFIPS)
  ELSIF (storage?(keyUsage(k)) or migrate?(keyUsage(k)))
     AND not RSA?(algoId(algoParms(k)))
  THEN OUT_Error(TPM_BAD_KEY_PROPERTY)
  ELSE LET DU1=tempAuthData,
           DM1=tempAuthData IN
       LET h=tpmCompositeHash((#select:=creationPCRSelect(PCRInfo(k)),
```

```
                                pcrValue:=s‘pcrs#)) IN
        LET encData=tpmStoreAsymkey(DU1,
                                    IF migratable(keyFlags(k)) THEN DM1
                                    ELSE s‘permData‘tpmProof ENDIF,
                                    pubDataDigest(encDat(k)),
                                    privKey(encDat(k)),
                                    clear),
            pcrs=tpmPCRInfoLong(s‘locality,
                                locAtRelease(PCRInfo(k)),
                                creationPCRSelect(PCRInfo(k)),
                                releasePCRSelect(PCRInfo(k)),
                                h,
                                digAtRelease(PCRInfo(k))) IN
        OUT_CreateWrapKey(tpmKey(s‘keyGenCnt,keyUsage(k),keyFlags(k),
                                 authDataUsage(k),algoParms(k),pcrs,key(p),
                                 encData,clear),
                          TPM_SUCCESS)
  ENDIF;

validateKeyConsistency(s:tpmAbsState,p,k:(tpmKey?)) : nat =
  IF migratable(keyFlags(k))=0 AND
     migrationAuth(encDat(k))/=tpmProof(permData(s))
  THEN 4
  ELSIF FIPS(permFlags(s)) AND (never?(authDataUsage(k)) OR
        legacy?(keyUsage(k)))
  THEN 3
  ELSE 0
  ENDIF;

validateLoadKey2(s:tpmAbsState,p,k:(tpmKey?)) : nat =
  IF not storage?(keyUsage(p))
  THEN 1
  ELSIF not(key(p)=wrappingKey(k))
  THEN 2
  ELSE CASES keyUsage(k) OF
         identity: IF migratable(keyFlags(k))=FALSE
                   THEN validateKeyConsistency(s,p,k)
                   ELSE 1
                   ENDIF,
         authChange: 1
         ELSE validateKeyConsistency(s,p,k)
       ENDCASES
  ENDIF;

loadKey2?(state:tpmAbsState,p,k:(tpmKey?)) : bool =
  validateLoadKey2(state,p,k)=0

loadKey2State(s:tpmAbsState,p,k:(tpmKey?)) : tpmAbsState =
  IF validateLoadKey2(s,p,k)=0
  THEN s with [‘keys:=loadKey(k,p,keys(s),pcrs(s))]
```

```
  ELSE s
  ENDIF;

loadKey2Out(s:tpmAbsState,parentH,inKey:(tpmKey?)) : tpmAbsOutput =
  LET num=validateLoadKey2(s,parentH,inKey) IN
    COND
    num=0 -> OUT_LoadKey2(inKey,TPM_SUCCESS),
    num=1 -> OUT_Error(TPM_INVALID_KEYUSAGE),
    num=2 -> OUT_CPUError(CPU_DECRYPT_ERROR),
    num=3 -> OUT_Error(TPM_NOTFIPS),
    ELSE -> OUT_Error(TPM_FAIL)
    ENDCOND;

getPubKey?(s:tpmAbsState,key:(tpmKey?)) : bool =
  never?(authDataUsage(key)) AND s`permFlags`readSRKPub
  AND (pcrIgnoredOnRead(keyFlags(key))
  OR dig(digAtRelease(PCRInfo(key)))=
        dig(digAtRelease(PCRInfo(key))) WITH [`pcrValue:=s`pcrs])

getPubKeyOut(s:tpmAbsState,k:(tpmKey?)) : tpmAbsOutput =
  IF not never?(authDataUsage(k))
  THEN OUT_Error(TPM_AUTHFAIL)
  ELSIF not(s`permFlags`readSRKPub)
  THEN OUT_Error(TPM_INVALID_KEYHANDLE)
  ELSIF not(pcrIgnoredOnRead(keyFlags(k)))
        AND dig(digAtRelease(PCRInfo(k)))/=
              dig(digAtRelease(PCRInfo(k))) WITH [pcrValue:=s`pcrs]
  THEN OUT_Error(TPM_WRONGPCRVAL)
  ELSE LET pubKey=tpmPubkey(key(k)) IN
       OUT_GetPubKey(pubKey,TPM_SUCCESS)
  ENDIF;

checkMigKeyAuth?(s:tpmAbsState,a:(tpmMigKeyAuth?)) : bool =
  digest(a)=tpmDigest(cons(key(a),
                       cons(scheme(a),
                                cons(s`permData`tpmProof,
                                     null))),
                      clear);

createMigBlob?(s:tpmAbsState,p:(tpmKey?),m:migrateScheme,a:(tpmMigKeyAuth?),
  e:tpmData) : bool =
  storage?(keyUsage(p)) AND checkMigKeyAuth?(s,a) AND
  CASES m OF
    migrate : TRUE,
    rewrap  : TRUE
    % ELSE FALSE
  ENDCASES

createMigBlobOut(s:tpmAbsState,p:(tpmKey?),migType:migrateScheme,
  mKeyAuth:(tpmMigKeyAuth?),encData:(tpmKey?)):tpmAbsOutput =
```

```
   LET d1:(tpmKey?)=decrypt(key(p),encData) IN
   IF not storage?(keyUsage(p))
   THEN OUT_Error(TPM_INVALID_KEYUSAGE)
   ELSIF not checkMigKeyAuth?(s,mKeyAuth)
   THEN OUT_Error(TPM_AUTHFAIL)
   ELSE CASES migType OF
          migrate: LET M1=tpmMigrateAsymkey(usageAuth(encDat(d1)),
                                            pubDataDigest(encDat(d1)),
                                            privKey(encDat(d1))) IN
                   OUT_CreateMigrationBlob(rand,M1,TPM_SUCCESS),
          rewrap : OUT_CreateMigrationBlob(0,
                                CASES encData OF
                                  tpmKey(v,u,f,d,g,p,w,e,c) :
                                     tpmKey(v,u,f,d,g,p,key(key(mKeyAuth)),e,c)
                                ENDCASES,
                                TPM_SUCCESS)
          %ELSE OUT_Error(TPM_BAD_PARAMETER)
ENDCASES
   ENDIF;

 convertMigBlobOut(s:tpmAbsState,parent:(tpmKey?),inData:(tpmMigrateAsymkey?),
   rand:int) : tpmAbsOutput =
   IF not storage?(keyUsage(parent))
   THEN OUT_Error(TPM_INVALID_KEYUSAGE)
   ELSE LET d1=decrypt(key(parent),inData) IN
        LET m1=inData,
        pHash:(tpmSecret?)=tempAuthData IN
        LET k1:KVAL=partPrivKey(m1) IN
        LET d2=tpmStoreAsymkey(usageAuth(m1),pHash,pubDataDigest(m1),k1,
                               encrypted(key(parent),clear)) IN
        OUT_ConvertMigrationBlob(d2,TPM_SUCCESS)
   ENDIF;

 authorizeMigKeyOut(s:tpmAbsState,migKey:(tpmKey?),migScheme:(tpmMigScheme?)):
   tpmAbsOutput =
   OUT_AuthorizeMigrationKey(
             tpmMigKeyAuth(migKey,migScheme,
                           tpmDigest(cons(migKey,cons(migScheme,
                                          cons(s`permData`tpmProof,null))),clear),
                           clear),
             TPM_SUCCESS);

 %% Decrypts the input packet (coming from TPM_CreateMigrationBlob) and then
 %% re-encrypts it with the input public key. The output would then be sent to
 %% TPM_ConvertMigrationBlob on the target TPM.
 migrateKeyOut(s:tpmAbsState,m,p:(tpmKey?),d:tpmData) : tpmAbsOutput =
   IF not migrate?(keyUsage(m))
   THEN OUT_Error(TPM_INVALID_KEYUSAGE)
   ELSE OUT_MigrateKey(encrypt(key(p),decrypt(key(m),d)),TPM_SUCCESS)
   ENDIF;
```

```
  sign?(s:tpmAbsState,key:(tpmKey?),areaToSign:tpmData) : bool =
    signing?(keyUsage(key)) OR legacy?(keyUsage(key));

  % Sign a blob if signing key isn't aik
  signOut(s:tpmAbsState,key:(tpmKey?),areaToSign:tpmData) : tpmAbsOutput=
    IF signing?(keyUsage(key)) OR legacy?(keyUsage(key))
    THEN OUT_Sign(sign(key(key),areaToSign),TPM_SUCCESS)
    ELSE OUT_Error(TPM_INVALID_KEYUSAGE)
    ENDIF;

  createEKPairState(s:tpmAbsState,r:(tpmNonce?),k:(tpmKey?)) : tpmAbsState =
    IF goodkey?(key(ek(s)))
    THEN s
    ELSE s WITH [`ek:=privateKey(k)
         ,`permFlags(CEKPUsed):=TRUE
,`permFlags(enableRevokeEK):=FALSE]
    ENDIF;

  % If ek doesn't exist, create a new ek from keyInfo
  createEKPairOut(s:tpmAbsState,r:(tpmNonce?),k:(tpmKey?)) : tpmAbsOutput =
    IF goodkey?(key(ek(s)))
    THEN OUT_Error(TPM_DISABLED_CMD)
    ELSE LET checksum=tpmDigest(cons(k,cons(r,null)),clear) IN
         OUT_CreateEndorsementKeyPair(k,checksum,TPM_SUCCESS)
    ENDIF;

  createRevEKState(s:tpmAbsState,antiReplay:(tpmNonce?),keyInfo:(tpmKey?),
    generateReset:bool,inputEKreset:(tpmNonce?)) : tpmAbsState =
    IF goodkey?(key(ek(s)))
    THEN s
    ELSE LET s1=createEKPairState(s,antiReplay,keyInfo) IN
         s1 WITH [`permFlags(enableRevokeEK):=TRUE
                 ,`permData(ekReset):=IF generateReset
                                      THEN tpmNonce(rand)
                                      ELSE inputEKreset
                                      ENDIF]
    ENDIF;

  createRevEKOut(s:tpmAbsState,r:(tpmNonce?),k:(tpmKey?),genReset:bool,
    i:(tpmNonce?)) : tpmAbsOutput =
    IF goodkey?(key(ek(s)))
    THEN OUT_Error(TPM_DISABLED_CMD)
    ELSE LET o1=createEKPairOut(s,r,k) IN
         CASES o1 OF
           OUT_CreateEndorsementKeyPair(e,c,m) :
               OUT_CreateRevocableEK(tpmPubkey(key(e)),c,
                                     IF genReset THEN tpmNonce(rand) ELSE i ENDIF,
                                     TPM_SUCCESS)
           ELSE o1
```

```
        ENDCASES
  ENDIF;

revokeTrust?(s:tpmAbsState,ekReset:(tpmNonce?)) : bool =
  s‘permFlags‘enableRevokeEK AND s‘permData‘ekReset=ekReset AND
   s‘stclearFlags‘physicalPresence

revokeTrustState(s:tpmAbsState,ekReset:(tpmNonce?)) : tpmAbsState =
  IF s‘permFlags‘enableRevokeEK AND ekReset(permData(s))=ekReset AND
     s‘stclearFlags‘physicalPresence
  THEN LET s1 = clear(s) IN
       s1 WITH [‘permFlags(nvLocked):=FALSE
               ,‘ek:=tpmKey(0,keyUsage(s‘ek),keyFlags(s‘ek),
                            authDataUsage(s‘ek),algoParms(s‘ek),
                            PCRInfo(s‘ek),wrappingKey(s‘ek),
                            encDat(s‘ek),crs(s‘ek))]
  ELSE s
  ENDIF;

revokeTrustOut(s:tpmAbsState,ekReset:(tpmNonce?)) : tpmAbsOutput =
  IF not s‘permFlags‘enableRevokeEK
  THEN OUT_Error(TPM_PERMANENTEK)
  ELSIF not s‘permData‘ekReset=ekReset
  THEN OUT_Error(TPM_AUTHFAIL)
  ELSIF not s‘stclearFlags‘physicalPresence
  THEN OUT_Error(TPM_BAD_MODE)
  ELSE OUT_RevokeTrust(TPM_SUCCESS)
  ENDIF;

readPubek?(s:tpmAbsState,n:(tpmNonce?)) : bool =
  readPubek(permFlags(s)) AND goodkey?(key(ek(s)));

readPubekOut(s:tpmAbsState,n:(tpmNonce?)) : tpmAbsOutput =
  IF not s‘permFlags‘readPubek
  THEN OUT_Error(TPM_DISABLED_CMD)
  ELSIF not goodkey?(key(ek(s)))
  THEN OUT_Error(TPM_NO_ENDORSEMENT)
  ELSE LET pubEK=publicKey(ek(s)) IN
       LET checksum=tpmDigest(cons(pubEK,cons(n,null)),clear) IN
       OUT_ReadPubek(pubEK,checksum,TPM_SUCCESS)
  ENDIF;

ownerReadInternalPubOut(s:tpmAbsState,k:(tpmKey?)) : tpmAbsOutput =
  IF key(k)=ekKeyVal
  THEN OUT_OwnerReadInternalPub(tpmPubkey(ekKeyVal),TPM_SUCCESS)
  ELSIF key(k)=srkKeyVal
  THEN OUT_OwnerReadInternalPub(tpmPubkey(srkKeyVal),TPM_SUCCESS)
  ELSE OUT_Error(TPM_BAD_PARAMETER)
  ENDIF;
```

```
makeIdentity?(s:tpmAbsState,k:(tpmKey?)) : bool=
  identity?(keyUsage(k)) AND not(migratable(keyFlags(k)))
  AND not(s`permFlags`FIPS AND never?(authDataUsage(k)))

makeIdentityState(s:tpmAbsState,CADig:(tpmDigest?),k:(tpmKey?)) :tpmAbsState=
  IF makeIdentity?(s,k)
  THEN genKeyState(s)
  ELSE s
  ENDIF;

setDigAtCreation(s:tpmAbsState,pcr:(tpmPCRInfoLong?)) : (tpmCompositeHash?) =
  LET digest:PCR_COMPOSITE=(# select:=creationPCRSelect(pcr),
                              pcrValue:=pcrs(s) #)
  IN tpmCompositeHash(digest);

makeIdentityLocality : LOCALITY; %placeholder

% Output a newly generated AIK.  Note that a simulates the new aik.
makeIdentityOut(s:tpmAbsState,CADig:(tpmDigest?),k:(tpmKey?)) : tpmAbsOutput =
  IF s`permFlags`FIPS AND never?(authDataUsage(k))
  THEN OUT_Error(TPM_NOTFIPS)
  ELSIF not(identity?(keyUsage(k))) OR migratable(keyFlags(k))
  THEN OUT_Error(TPM_INVALID_KEYUSAGE)
  ELSE LET a1=tempAuthData IN
       LET pcr=tpmPCRInfoLong(makeIdentityLocality,
                              locAtRelease(PCRInfo(k)),
                              creationPCRSelect(PCRInfo(k)),
                              releasePCRSelect(PCRInfo(k)),
                              setDigAtCreation(s,PCRInfo(k)),
                              digAtRelease(PCRInfo(k))),
           encData=tpmStoreAsymkey(a1,s`permData`tpmProof,
                                   pubDataDigest(encDat(k)),
                                   privKey(encDat(k)),clear) IN
       LET idKey=tpmKey(keyGenCnt(s),keyUsage(k),keyFlags(k),
                        authDataUsage(k),algoParms(k),pcr,wrappingKey(s`srk),
                        encData,clear) IN
       LET idBind=tpmIdContents(CADig,idKey,signed(private(idKey),clear)) IN
       OUT_MakeIdentity(idKey,idBind,TPM_SUCCESS)
  ENDIF;

activateIdentity?(s:tpmAbsState,i:(tpmKey?),b:(activateIdentityBlob?)) : bool=
  LET h1=tpmDigest(cons(tpmPubkey(key(i)),null),clear),
      b1=decrypt(private(ekVal),b) IN
  identity?(keyUsage(i)) AND
  CASES b1 OF
    tpmAsymCAContents(k,d,crs) : h1=d AND not(encrypted?(crs)),
    tpmEKBlob(blob,crs) :
            CASES blob OF
              tpmEKBlobActivate(k,d,p) :
```

```
              LET C1=tpmCompositeHash((#select:=pcrSelect(p),
                                       pcrValue:=pcrs(s) #)) IN
              h1=d AND (null?(pcrSelect(p)) OR C1=digAtRelease(p))
              AND member(s`locality,locAtRelease(p))
            ELSE FALSE
          ENDCASES
    ELSE FALSE
  ENDCASES;

% Retrieve a key if a can be installed
% The cmd assumes the availability of the priv key associated with th identity
% The cmd will verify the association between the keys during the process.
% The cmd will decrypt the input blob and extract the sess key and verify
% the connection between the public and private keys. p 157
activateIdentityOut(s:tpmAbsState,idKey:(tpmKey?),b:(activateIdentityBlob?)):
  tpmAbsOutput =
  IF not identity?(keyUsage(idKey))
  THEN OUT_Error(TPM_BAD_PARAMETER)
  ELSE LET h1=tpmDigest(cons(tpmPubkey(key(idKey)),null),clear),
           b1=decrypt(private(ekVal),b) IN
       CASES b1 OF
         tpmAsymCAContents(k,d,crs) :
               IF h1=d and not(encrypted?(crs))
               THEN OUT_ActivateIdentity(k,TPM_SUCCESS)
               ELSE OUT_CPUError(CPU_DECRYPT_ERROR)
               ENDIF,
         tpmEKBlob(blob,crs) :
               CASES blob OF
                 tpmEKBlobActivate(k,d,p) :
                   LET C1=tpmCompositeHash((#select:=pcrSelect(p),
                                            pcrValue:=pcrs(s) #)) IN
                   IF h1/=d
                   THEN OUT_Error(TPM_BAD_PARAMETER)
                   ELSIF not(null?(pcrSelect(p))) AND C1/=digAtRelease(p)
                   THEN OUT_Error(TPM_WRONGPCRVAL)
                   ELSIF not member(s`locality,locAtRelease(p))
                   THEN OUT_Error(TPM_BAD_LOCALITY)
                   ELSE OUT_ActivateIdentity(k,TPM_SUCCESS)
                   ENDIF
                 ELSE OUT_Error(TPM_BAD_TYPE)
               ENDCASES
         ELSE OUT_Error(TPM_SUCCESS) % Should be unreachable case.
       ENDCASES
  ENDIF;

% For PCRS hashing is pcrsExtend instead of tpmDigest
extend?(s:tpmAbsState,pcrNum:PCRINDEX,inDigest:HV) : bool =
  LET L1=s`stanyFlags`localityModifier,
      P1=pcrExtendLocal(s`permData`pcrAttrib(pcrNum)) IN
      (0<=pcrNum<=23) AND member(L1,P1);
```

```
%% Extend operation on TPM state
extendState(s:tpmAbsState,pcrNum:PCRINDEX,inDigest:HV) : tpmAbsState =
  IF extend?(s,pcrNum,inDigest)
  THEN s WITH [`pcrs := pcrsExtend(pcrs(s),pcrNum,inDigest)]
  ELSE s
  ENDIF;

extendOut(s:tpmAbsState,pcrNum:PCRINDEX,inDigest:HV) : tpmAbsOutput =
  LET L1=s`stanyFlags`localityModifier,
      P1=pcrExtendLocal(s`permData`pcrAttrib(pcrNum)),
      H1=pcrsExtend(s`pcrs,pcrNum,inDigest) IN
  IF pcrNum > 23 OR pcrNum < 0
  THEN OUT_Error(TPM_BADINDEX)
  ELSIF not member(L1,P1)
  THEN OUT_Error(TPM_BAD_LOCALITY)
  ELSIF s`permFlags`disable OR s`stclearFlags`deactivated
  THEN OUT_Extend(reset,TPM_SUCCESS)
  ELSE OUT_Extend(extend(s`pcrs(pcrNum),inDigest),TPM_SUCCESS)
  ENDIF   ;

% Output PCR of given index
pcrReadOut(s:tpmAbsState,ind:PCRINDEX) : tpmAbsOutput =
  IF ind > 23 OR ind < 0
  THEN OUT_Error(TPM_BADINDEX)
  ELSE LET p=s`pcrs IN
       OUT_PCRRead(p(ind),TPM_SUCCESS)
  ENDIF;

% Output PCRs from a state as quote
quote?(k:(tpmKey?)) : bool =
  signing?(keyUsage(k)) OR identity?(keyUsage(k)) OR legacy?(keyUsage(k))

quoteOut(s:tpmAbsState,k:(tpmKey?),extDat:(tpmNonce?),p:PCR_SELECTION) :
 tpmAbsOutput =
  IF not quote?(k)
  THEN OUT_Error(TPM_INVALID_KEYUSAGE)
  ELSE LET H1=tpmCompositeHash((#select:=p,pcrValue:=s`pcrs#)),
           pcrData=getPCRs(s`pcrs,p) IN
OUT_Quote(pcrData,tpmQuote(H1,extDat,signed(private(k),clear)),TPM_SUCCESS)
  ENDIF;

validatePCRVals(s:tpmAbsState,pcrSelect:PCR_SELECTION) : RECURSIVE nat =
  LET L1=s`stanyFlags`localityModifier,
   sum:nat=0 IN
  CASES pcrSelect OF
    null : 0,
    cons(x,xs) : IF not pcrReset(s`permData`pcrAttrib(x))
                 THEN 1
                 ELSIF not member(L1,pcrResetLocal(s`permData`pcrAttrib(x)))
```

```
                THEN 2
                ELSE validatePCRVals(s,xs)
                ENDIF
  ENDCASES
  measure pcrSelect by <<

validatePCRs : LEMMA
  FORALL (s:tpmAbsState,select:PCR_SELECTION) :
  LET a=validatePCRVals(s,select) IN
  a=0 OR a=1 or a=2

check_validatePcrs: THEOREM
LET s:tpmAbsState=tpmStartup WITH [‘permData(pcrAttrib)(6):=
                                        (# pcrReset:=true,
                                        pcrResetLocal:=cons(2,null),
                                        pcrExtendLocal:=allLocs#)],
    p:PCR_SELECTION=cons(5,cons(6,null)) IN
  validatePCRVals(s,p)=2;

pcrResetState(s:tpmAbsState,pcrSelect:PCR_SELECTION) : tpmAbsState  =
  IF null?(pcrSelect)
  THEN s
  ELSE LET num=validatePCRVals(s,pcrSelect) IN
       COND
         num=0 -> s WITH [‘pcrs:=pcrsResetSelection(s‘pcrs,pcrSelect)],
         ELSE -> s
       ENDCOND
  ENDIF;

pcrResetOut(s:tpmAbsState,pcrSelect:PCR_SELECTION) : tpmAbsOutput =
  IF null?(pcrSelect)
  THEN OUT_Error(TPM_INVALID_PCR_INFO)
  ELSE LET num=validatePCRVals(s,pcrSelect) IN
       COND
         num=0 -> OUT_PCR_Reset(TPM_SUCCESS),
         num=1 -> OUT_Error(TPM_NOTRESETABLE),
         num=2 -> OUT_Error(TPM_NOTLOCAL),
         ELSE -> OUT_Error(TPM_SUCCESS)
       ENDCOND
  ENDIF;

dataBindOut(s:tpmAbsState,k:(tpmKey?),d:tpmData) : tpmAbsOutput =
  OUT_Data_Bind(tpmBoundData(d,encrypted(key(k),clear)),CPU_SUCCESS);

certify?(aik:(tpmKey?),certReq:(tpmIdContents?)) : bool =
  LET d=checkSig(key(aik),certReq) IN
    d AND tpmDigest?(digest(certReq));

certState(s:tpmAbsState,aik:(tpmKey?),certReq:(tpmIdContents?)) : tpmAbsState=
  IF certify?(aik,certReq)
```

```
  THEN genKeyState(s)
  ELSE s
  ENDIF;

certOut(s:tpmAbsState,aik:(tpmKey?),certReq:(tpmIdContents?)) : tpmAbsOutput =
  IF certify?(aik,certReq)
  THEN OUT_Certify(aik,
                   tpmAsymCAContents(tpmSessKey(keyGenCnt(s),clear),
                                     digest(certReq),
                                     encrypted(key(ekVal),clear)),
                   CPU_SUCCESS)
  ELSE OUT_CPUError(CPU_DECRYPT_ERROR)
  ENDIF;

readOut(s:tpmAbsState,i:nat) : tpmAbsOutput =
  s`memory(i);

revokeKeyState(s:tpmAbsState,k:(tpmKey?)) : tpmAbsState =
  s WITH [`keys := revokeKey(k,keys(s))];


restoreState(s:tpmAbsState) : tpmAbsState =
  IF valid?(restore(s))
  THEN LET rs=restore(s) IN
       (# restore := rs
        , memory := memory(s)
        , ek := ek(rs)
        , srk := srk(rs)
        , pcrs := pcrs(rs)
        , keys := keys(rs)
        , keyGenCnt:=keyGenCnt(rs)
        , locality := 3
        , permFlags := permFlags(rs)
        , permData := permData(rs)
        , stanyFlags := stanyFlags(rs)
        , stanyData := stanyData(rs)
        , stclearFlags := stclearFlags(rs)
        , stclearData := stclearData(rs)
        #)
  ELSE s
  ENDIF;

% deactivate by going back to init.  Not sure this is correct.
deactivateState(s:tpmAbsState) : tpmAbsState =
  s WITH [`stanyFlags(postInitialize) := TRUE];

% save a value to external memory
saveToMemState(s:tpmAbsState,i:nat,v:tpmAbsOutput) : tpmAbsState =
  s WITH [`memory := updateLoc(memory(s),i,v)];
```

```
%% What we want to generate with each command is a pair of type
%% [tpmAbsOut,tpmAbsState].  executeCom generates the the state
%% from a TPM state and command while the upcoming outputCom
%% generates the corresponding output.  Thus, to execute any
%% operation, both executeCom and outputCom must be used

%% Run if TPM_Init has run but TPM_Startup has not
executeComPostInit(s:(afterInit?),c:tpmAbsInput) : tpmAbsState =
  CASES c OF
    ABS_Startup(t) : CASES t OF
                       TPM_ST_CLEAR : tpmStartup,
                       TPM_ST_STATE : restoreState(s),
                       TPM_ST_DEACTIVATED : deactivateState(s)
                     ENDCASES
    ELSE s
  ENDCASES;

%% Run if ABS_Startup has run after TPM_Init
executeComStartup(s:(afterStartup?),c:tpmAbsInput) : tpmAbsState =
  CASES c OF
    ABS_Init : tpmPostInit,
    ABS_SaveState : saveState(s),
    ABS_SetOwnerInstall(state) : setOwnerInstallState(s,state),
    ABS_OwnerSetDisable(d) : ownerSetDisableState(s,d),
    ABS_PhysicalEnable : physicalEnableState(s),
    ABS_PhysicalDisable : physicalDisableState(s),
    ABS_PhysicalSetDeactivated(st) : physicalSetDeactivatedState(s,st),
    ABS_SetTempDeactivated : setTempDeactivatedState(s),
    ABS_SetOperatorAuth(opAuth) : setOperatorAuthState(s,opAuth),
    ABS_TakeOwnership(srk) : takeOwnershipState(s,srk),
    ABS_OwnerClear: ownerClearState(s),
    ABS_ForceClear: forceClearState(s),
    ABS_DisableOwnerClear: disableOwnerClearState(s),
    ABS_DisableForceClear: disableForceClearState(s),
    ABS_PhysicalPresence(p): physicalPresenceState(s,p),
    ABS_ResetEstablishmentBit: resetEstablishmentBitState(s),
    ABS_CreateWrapKey(p,k) : createWrapKeyState(s,p,k),
    ABS_LoadKey2(p,k): loadKey2State(s,p,k),
    ABS_CreateEndorsementKeyPair(n,k) : createEKPairState(s,n,k),
    ABS_CreateRevocableEK(r,k,g,i) : createRevEKState(s,r,k,g,i),
    ABS_RevokeTrust(r) : revokeTrustState(s,r),
    ABS_MakeIdentity(d,k) : makeIdentityState(s,d,k),
    ABS_Extend(n,d) : extendState(s,n,d),
    ABS_PCR_Reset(p) : pcrResetState(s,p),
    ABS_certify(k,cr) : certState(s,k,cr),
    ABS_senter : changeLocalityState(extendState(pcrsResetSenterState(s),
                                     0,hash(sinit))),
    ABS_sinit : changeLocalityState(extendState(s,0,hash(mle))),
    ABS_save(i,v) : saveToMemState(s,i,v)
    ELSE s
```

```
ENDCASES;

%% Execute a command on state generating a new state.  Commands execution
%% should be blocked if startup has not run and postInitialize=true.
executeCom(s:tpmAbsState,c:tpmAbsInput) : tpmAbsState =
  IF afterInit?(s)
  THEN executeComPostInit(s,c)
  ELSE executeComStartup(s,c)
  ENDIF;

%% Run if TPM_Init has run but TPM_Startup has not
outputComPostInit(s:(afterInit?),c:tpmAbsInput) : tpmAbsOutput =
  OUT_Error(TPM_SUCCESS);

%% Generate output from a command and state
%% Run if ABS_Startup has run after TPM_Init
outputComStartup(s:(afterStartup?),c:tpmAbsInput) : tpmAbsOutput =
  CASES c OF
    ABS_SetOwnerInstall(state) : setOwnerInstallOut(s,state),
    ABS_OwnerSetDisable(d) : ownerSetDisableOut(s,d),
    ABS_PhysicalEnable : physicalEnableOut(s),
    ABS_PhysicalDisable : physicalDisableOut(s),
    ABS_PhysicalSetDeactivated(st) : physicalSetDeactivatedOut(s,st),
    ABS_SetTempDeactivated : setTempDeactivatedOut(s),
    ABS_SetOperatorAuth(a) : setOperatorAuthOut(s,a),
    ABS_TakeOwnership(srk) : takeOwnershipOut(s,srk),
    ABS_OwnerClear : ownerClearOut(s),
    ABS_ForceClear : forceClearOut(s),
    ABS_DisableOwnerClear : disableOwnerClearOut(s),
    ABS_DisableForceClear : disableForceClearOut(s),
    ABS_PhysicalPresence(p) : physicalPresenceOut(s,p),
    ABS_ResetEstablishmentBit : resetEstablishmentBitOut(s),
    ABS_Seal(k,p,data) : sealOut(s,k,p,data),
    ABS_Unseal(d,k) : unsealOut(s,d,k),
    ABS_UnBind(k,d) : unBindOut(s,k,d),
    ABS_Data_Bind(k,d) : dataBindOut(s,k,d),
    ABS_CreateWrapKey(parent,k) : createWrapKeyOut(s,parent,k),
    ABS_LoadKey2(p,k) : loadKey2Out(s,p,k),
    ABS_GetPubKey(k): getPubKeyOut(s,k),
    ABS_CreateMigrationBlob(pk,m,a,e) : createMigBlobOut(s,pk,m,a,e),
    ABS_ConvertMigrationBlob(p,d,r) : convertMigBlobOut(s,p,d,r),
    ABS_AuthorizeMigrationKey(mk,ms) : authorizeMigKeyOut(s,mk,ms),
    ABS_MigrateKey(mk,pubKey,data) : migrateKeyOut(s,mk,pubKey,data),
    ABS_Sign(k,a) : signOut(s,k,a),
    ABS_CreateEndorsementKeyPair(n,k) : createEKPairOut(s,n,k),
    ABS_CreateRevocableEK(r,k,g,i) : createRevEKOut(s,r,k,g,i),
    ABS_RevokeTrust(r) : revokeTrustOut(s,r),
    ABS_ReadPubek(n) : readPubekOut(s,n),
    ABS_OwnerReadInternalPub(k) : ownerReadInternalPubOut(s,k),
    ABS_MakeIdentity(d,k) : makeIdentityOut(s,d,k),
```

```
    ABS_ActivateIdentity(i,b) : activateIdentityOut(s,i,b),
    ABS_Extend(n,d) : extendOut(s,n,d),
    ABS_PCRRead(ind) : pcrReadOut(s,ind),
    ABS_Quote(k,n,pm) : quoteOut(s,k,n,pm),
    ABS_PCR_Reset(p) : pcrResetOut(s,p),
    ABS_certify(aik,cr) : certOut(s,aik,cr),
    ABS_read(i) : readOut(s,i)
  ELSE OUT_Error(TPM_SUCCESS)
ENDCASES;

%% Execute a command on state generating a new state.  Commands execution
%% should be blocked if startup has not run and postInitialize=true.
outputCom(s:tpmAbsState,c:tpmAbsInput) : tpmAbsOutput =
  IF afterInit?(s)
     THEN outputComPostInit(s,c)
     ELSE outputComStartup(s,c)
  ENDIF;


%% CPU Command Definitions

%% Requires use of bind
CPU_saveOutput(i:nat) : [tpmAbsOutput -> State] =
    useOutputStateOutput(
              (LAMBDA (a:tpmAbsOutput):
                 (LAMBDA (s:tpmAbsState):
                     executeCom(s,ABS_save(i,a)))),
              (LAMBDA (a:tpmAbsOutput):
                 (LAMBDA (s:tpmAbsState):
                     a)));

CPU_read(i:nat) : State =
   output(LAMBDA (s:tpmAbsState): outputCom(s,ABS_read(i)));

%% Call SENTER
CPU_senter : State =
  modify(OUT_Error(TPM_SUCCESS),
         (LAMBDA (s:tpmAbsState):executeCom(s,ABS_senter)));

%% Call sinit for initial measurements
CPU_sinit : State =
  modify(OUT_Error(TPM_SUCCESS),
         (LAMBDA (s:tpmAbsState):executeCom(s,ABS_sinit)));

CPU_BuildQuoteFromMem(q,idcont:nat):State =
  output(LAMBDA (s:tpmAbsState) :
     LET theMem = memory(s) IN
     IF OUT_Quote?(theMem(q)) AND tpmQuote?(sig(theMem(q)))
        AND OUT_MakeIdentity?(theMem(idcont))
     THEN  OUT_FullQuote(sig(theMem(q)),idBinding(theMem(idcont)),CPU_SUCCESS)
```

```
     ELSE OUT_CPUError(CPU_QUOTE_ERROR)
     ENDIF);

gen_quote : THEOREM
 FORALL (state:(afterStartup?),x,y:nat) :
  LET (a,s) = runState(CPU_BuildQuoteFromMem(x,y))(state) IN
  IF OUT_Quote?(s`memory(x)) AND
     tpmQuote?(sig(s`memory(x))) AND
     OUT_MakeIdentity?(s`memory(y))
  THEN a=OUT_FullQuote(sig(s`memory(x)),idBinding(s`memory(y)),CPU_SUCCESS)
  ELSE a=OUT_CPUError(CPU_QUOTE_ERROR)
  ENDIF
  AND s=state;

%% CA Command Definitions
%% Invoke the certification authority
CA_certify(aik:(tpmKey?),cr:(tpmIdContents?)) : State =
  modifyOutput(
          (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_certify(aik,cr))),
          (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_certify(aik,cr))));

%% Assure the CA generates an appropriate cert from the AIK and Ek
%% proved - Wed Jul 11 13:14:11 EDT 2012
gen_cert: THEOREM
 FORALL (state:(afterStartup?),aik:(tpmKey?),id:(tpmIdContents?)) :
  LET (a,s) = runState(CA_certify(aik,id))(state) IN
  IF certify?(aik,id)
  THEN a=OUT_Certify(aik,
                     tpmAsymCAContents(tpmSessKey(keyGenCnt(state),clear),
                                       digest(id),
                                       encrypted(key(ekVal),clear)),
                     CPU_SUCCESS) AND
       s = state WITH [`keyGenCnt := keyGenCnt(state)+1]
  ELSE a=OUT_CPUError(CPU_DECRYPT_ERROR) AND
       s=state
  ENDIF

%% TPM Command Definitions - Note that all commands used by the TPM
%% have the TPM suffix to distinguish from operations on tpm state.
%% Commands thus far are defined as state transforming or output
%% generating.  This does not exclude the eventuality of doing both.

%% No-op - not a real TPM command
TPM_Noop : [tpmAbsOutput -> State] =
  (LAMBDA (a:tpmAbsOutput): state((LAMBDA(s:tpmAbsState) : (a,s))));

%% TPM is off - not a real TPM command
TPM_Off : State = put(OUT_Nothing,tpmUnknown);

%% Power up the TPM and hit the TPM_Init signal
```

```
TPM_Init : State = put(OUT_Init(TPM_SUCCESS),tpmPostInit);

init_post : THEOREM
  FORALL (state:tpmAbsState) :
    LET (a,s) = runState(TPM_Init)(state) IN
        a=OUT_Init(TPM_SUCCESS) AND
        s=tpmPostInit

%% Save the TPM state in preparation for restore at startup
TPM_SaveState : State =
  modify(OUT_SaveState(TPM_SUCCESS),
          (LAMBDA (s:tpmAbsState):executeCom(s,ABS_SaveState)))

%% Save state actually saves the correct information.
%% proved - Fri Sep 14 11:20:25 CDT 2012
save_state_post : THEOREM
  FORALL (s0:(afterStartup?)) :
    LET (a,s) = runState(TPM_SaveState)(s0) IN
    LET save = restore(s) IN
    valid?(save)
    AND keys(save) = keys(s0)
    AND ek(save) = ek(s0)
    AND srk(save) = srk(s0)
    AND FORALL (i:PCRINDEX) :
          IF pcrReset(pcrAttrib(permData(save))(i))
          THEN pcrs(save)(i) = resetOne
          ELSE pcrs(save)(i) = pcrs(s0)(i)
          ENDIF
    AND permFlags(save) = permFlags(s0)
    AND permData(save) = permData(s0)
    AND a=OUT_SaveState(TPM_SUCCESS)

%% Start up the TPM after TPM_Init
TPM_Startup(st:TPM_STARTUP_TYPE) : State =
  modify(OUT_Startup(TPM_SUCCESS),
          (LAMBDA (s:tpmAbsState):executeCom(s,ABS_Startup(st))));

%% TPM_Startup post condition covering three major cases: clear, restore
%% state and deactivate.
%% proved - Sat Sep 15 09:58:36 CDT 2012
startup_post : THEOREM
  FORALL (s0:(afterInit?), f:TPM_STARTUP_TYPE) :
    LET (a,s) = runState(TPM_Startup(f))(s0) IN
    LET save = restore(s) IN
    CASES f OF
      TPM_ST_CLEAR : s = tpmStartup,
      TPM_ST_STATE : valid?(save) AND wellFormedRestore?(save) =>
                       keys(save) = keys(s)
                       AND ek(save) = ek(s)
                       AND srk(save) = srk(s)
```

```
                        AND FORALL (i:PCRINDEX) :
                              IF pcrReset(pcrAttrib(permData(save))(i))
                              THEN pcrs(save)(i) = resetOne
                              ELSE pcrs(save)(i) = pcrs(s)(i)
                              ENDIF
                        AND permFlags(save) = permFlags(s)
                        AND permData(save) = permData(s),
      TPM_ST_DEACTIVATED : postInitialize(stanyFlags(s))
    ENDCASES
    AND a=OUT_Startup(TPM_SUCCESS);

%% Prove that the only command that can run and do anyting folling a
%% TPM_Init is TPM_Startup.  If anything else tries to run, nothing happens.
%% Note the use of an "anything" command in the first command sequence.
startup_after_init: THEOREM
  FORALL (s:tpmAbsState,a:tpmAbsOutput,c:tpmAbsInput) :
    NOT ABS_Startup?(c) =>
      LET (a0,s0) = runState(TPM_Init >> state(LAMBDA (s:tpmAbsState) :
                                                (a,executeCom(s,c))))(s) IN
      LET (a1,s1) = runState(TPM_Init)(s) IN
      s0=s1;

%% Prove that senter after power up results in reset PCRs followed by sinit
%% measurement.
%% proved - Tue Jun 12 15:31:12 CDT 2012
resetMonad: THEOREM
  FORALL (hv:HV, state:tpmAbsState) :
    LET (a,s) = runState(
                  TPM_Init
                  >> TPM_Startup(TPM_ST_CLEAR)
                  >> CPU_senter
                  >> CPU_sinit)
                  (state) IN
    s = changeLocalityState(
           changeLocalityState(
              extendState(
                 extendState(pcrsResetSenterState(tpmStartup),0,hash(sinit)),
                 0,hash(mle)))) AND
    a = OUT_Error(TPM_SUCCESS);

% grind, decompose-equality
unique_error: LEMMA
  FORALL (b0,b1:ReturnCode) : OUT_Error(b0)=OUT_Error(b1) iff b0=b1;

TPM_SetOwnerInstall(state:bool) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_SetOwnerInstall(state))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_SetOwnerInstall(state))));

set_owner_install_post: THEOREM FORALL (state:(afterStartup?),b:bool) :
```

```
  LET (a,s) = runState(TPM_SetOwnerInstall(b))(state) IN
  IF state`permFlags`ownership
  THEN a=OUT_SetOwnerInstall(TPM_SUCCESS) AND s=state
  ELSIF state`stclearFlags`physicalPresence
  THEN a=OUT_SetOwnerInstall(TPM_SUCCESS) AND
       s=state WITH [`permFlags`ownership:=b]
  ELSE a=OUT_Error(TPM_BAD_PRESENCE) AND s=state
  ENDIF;


TPM_OwnerSetDisable(d:bool) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_OwnerSetDisable(d))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_OwnerSetDisable(d))));

owner_set_disable_post:THEOREM FORALL (state:(afterStartup?),b:bool):
  LET (a,s) = runState(TPM_OwnerSetDisable(b))(state) IN
  a=OUT_OwnerSetDisable(TPM_SUCCESS) AND
  s=state WITH [`permFlags`disable:=b];

TPM_PhysicalEnable : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_PhysicalEnable)),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_PhysicalEnable)));

physical_enable_post: THEOREM FORALL (state:(afterStartup?)) :
  LET (a,s) = runState(TPM_PhysicalEnable)(state) IN
  IF not state`stclearFlags`physicalPresence
  THEN a=OUT_Error(TPM_BAD_PRESENCE) AND s=state
  ELSE a=OUT_PhysicalEnable(TPM_SUCCESS) AND
       s=state WITH [`permFlags`disable:=FALSE]
  ENDIF;

TPM_PhysicalDisable : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_PhysicalDisable)),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_PhysicalDisable)));

physical_disable_post: THEOREM FORALL (state:(afterStartup?)) :
  LET (a,s) = runState(TPM_PhysicalDisable)(state) IN
  IF not state`stclearFlags`physicalPresence
  THEN a=OUT_Error(TPM_BAD_PRESENCE) AND s=state
  ELSE a=OUT_PhysicalDisable(TPM_SUCCESS) AND
       s=state WITH [`permFlags`disable:=TRUE]
  ENDIF;

TPM_PhysicalSetDeactivated(st:bool) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_PhysicalSetDeactivated(st))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_PhysicalSetDeactivated(st))));
```

```
physical_set_deactivated_post:THEOREM FORALL (state:(afterStartup?),st:bool) :
  LET (a,s) = runState(TPM_PhysicalSetDeactivated(st))(state) IN
  IF not state‘stclearFlags‘physicalPresence
  THEN a=OUT_Error(TPM_BAD_PRESENCE) AND s=state
  ELSE a=OUT_PhysicalSetDeactivated(TPM_SUCCESS) AND
       s=state WITH [‘permFlags‘disable:=st]
  ENDIF;

TPM_SetTempDeactivated : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_SetTempDeactivated)),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_SetTempDeactivated)));

set_temp_deactivated_post: THEOREM FORALL (state:(afterStartup?)) :
  LET (a,s) = runState(TPM_SetTempDeactivated)(state) IN
  IF not state‘permFlags‘operator
  THEN a=OUT_Error(TPM_NOOPERATOR) AND s=state
  ELSIF not state‘stclearFlags‘physicalPresence
  THEN a=OUT_Error(TPM_BAD_PRESENCE) AND s=state
  ELSE a=OUT_SetTempDeactivated(TPM_SUCCESS) AND
       s=state WITH [‘stclearFlags‘deactivated:=TRUE]
  ENDIF;

TPM_SetOperatorAuth(opAuth:(tpmSecret?)) : State =
   modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_SetOperatorAuth(opAuth))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_SetOperatorAuth(opAuth))));

set_operator_auth_post:THEOREM FORALL(state:(afterStartup?),op:(tpmSecret?)) :
  LET (a,s) = runState(TPM_SetOperatorAuth(op))(state) IN
  IF not state‘stclearFlags‘physicalPresence
  THEN a=OUT_Error(TPM_BAD_PRESENCE) AND s=state
  ELSE a=OUT_SetOperatorAuth(TPM_SUCCESS) AND
       s=state WITH [‘permData‘operatorAuth:=op
                    ,‘permFlags‘operator:=TRUE]
  ENDIF;

%% Establish SRK
TPM_TakeOwnership(srk:(tpmKey?)) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_TakeOwnership(srk))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_TakeOwnership(srk))));

take_ownership_post: THEOREM FORALL (state:(afterStartup?),srk:(tpmKey?)) :
  LET (a,s) = runState(TPM_TakeOwnership(srk))(state) IN
  IF takeOwnership?(state,srk)
  THEN LET A2=tempAuthData IN
       LET asymkey=tpmStoreAsymkey(A2,migrationAuth(encDat(srk)),
                           pubDataDigest(encDat(srk)),privKey(encDat(srk)),
```

```
                          crs(encDat(srk))) IN
      LET K1=tpmKey(key(srk),keyUsage(srk),keyFlags(srk),authDataUsage(srk),
                 algoParms(srk),PCRInfo(srk),wrappingKey(srk),asymkey,crs(srk)) IN
      a=OUT_TakeOwnership(K1,TPM_SUCCESS) AND
      s=state WITH [`srk:=K1
                   ,`permFlags(readPubek):=FALSE]
  ELSIF i(state`permData`ownerAuth)/=INVALIDAUTH
  THEN a=OUT_Error(TPM_OWNER_SET) AND s=state
  ELSIF not state`permFlags`ownership
  THEN a=OUT_Error(TPM_INSTALL_DISABLED) AND s=state
  ELSIF not goodkey?(key(state`ek))
  THEN a=OUT_Error(TPM_NO_ENDORSEMENT) AND s=state
  ELSIF not storage?(keyUsage(srk))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF migratable(keyFlags(srk))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF not RSA?(algoId(algoParms(srk)))
  THEN a=OUT_Error(TPM_BAD_KEY_PROPERTY) AND s=state
  ELSIF s`permFlags`FIPS AND never?(authDataUsage(srk))
  THEN a=OUT_Error(TPM_NOTFIPS) AND s=state
  ELSE a=OUT_Error(TPM_SUCCESS) AND s=state
  ENDIF;

take_ownership_post2: THEOREM FORALL (state:(afterStartup?),srk:(tpmKey?)) :
  LET (a,s) = runState(TPM_TakeOwnership(srk))(state) IN
  not (takeOwnership?(state,srk)) =>
    not (a=OUT_Error(TPM_SUCCESS))

% To be used with commands that use clear command:
%  TPM_OwnerClear, TPM_ForceClear, TPM_RevokeTrust
clear_post(s,state:(afterStartup?),a,af,at:tpmAbsOutput,p:bool) : bool =
  s`permData`pcrAttrib = state`permData`pcrAttrib AND
  s`permData`ekReset = state`permData`ekReset AND
  IF p
  THEN a=at AND s=state
  ELSE a=af AND keys(s) = emptyset AND
       s`permData`ownerAuth = tpmSecret(INVALIDAUTH) AND
       s`permData`tpmProof = tpmSecret(INVALIDPROOF) AND
       s`permData`operatorAuth = tpmSecret(INVALIDAUTH) AND
       s`permFlags = state`permFlags WITH
                          [`disable:=disableDef
                          ,`deactivated:=deactivatedDef
                          ,`readPubek:=readPubekDef
                          ,`disableOwnerClear:=disableOwnerClearDef
                          ,`disableFullDALogicInfo:=disableFullDALogicInfoDef
                          ,`allowMaintenance:=allowMaintenanceDef
                          ,`readSRKPub:=readSRKPubDef
                          ,`ownership:=TRUE
                          ,`operator:=FALSE
                          ,`maintenanceDone:=FALSE]
```

```
ENDIF;

 TPM_OwnerClear : State =
    modifyOutput(
           (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_OwnerClear)),
           (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_OwnerClear)));

 owner_clear_post: THEOREM FORALL (state:(afterStartup?)) :
   LET(a,s) = runState(TPM_OwnerClear)(state) IN
   clear_post(s,state,
              a,OUT_OwnerClear(TPM_SUCCESS),OUT_Error(TPM_CLEAR_DISABLED),
              state`permFlags`disableOwnerClear);

 TPM_ForceClear : State =
   modifyOutput(
     (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_ForceClear)),
     (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_ForceClear)));

 force_clear_post: THEOREM FORALL (state:(afterStartup?)) :
   LET(a,s) = runState(TPM_ForceClear)(state) IN
   IF s`stclearFlags`physicalPresence
   THEN clear_post(s,state,
                   a,OUT_ForceClear(TPM_SUCCESS),OUT_Error(TPM_CLEAR_DISABLED),
                   state`stclearFlags`disableForceClear)
   ELSE a=OUT_Error(TPM_BAD_PRESENCE) and s=state
   ENDIF;

 TPM_DisableOwnerClear : State =
    modifyOutput(
      (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_DisableOwnerClear)),
      (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_DisableOwnerClear)));

 disable_owner_clear_post: THEOREM FORALL (state:(afterStartup?)) :
   LET(a,s) = runState(TPM_DisableOwnerClear)(state) IN
   s=state with [`permFlags`disableOwnerClear:=TRUE] AND
   a=OUT_DisableOwnerClear(TPM_SUCCESS);

 TPM_DisableForceClear : State =
    modifyOutput(
      (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_DisableForceClear)),
      (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_DisableForceClear)));

 disable_force_clear_post: THEOREM FORALL (state:(afterStartup?)) :
   LET(a,s) = runState(TPM_DisableForceClear)(state) IN
   s`stclearFlags`disableForceClear = TRUE AND
   a = OUT_DisableForceClear(TPM_SUCCESS);

 TSC_PhysicalPresence(p:PHYSPRES) : State =
    modifyOutput(
      (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_PhysicalPresence(p))),
```

```
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_PhysicalPresence(p))));

physical_presence_post: THEOREM FORALL (state:(afterStartup?),p:PHYSPRES) :
  LET(a,s) = runState(TSC_PhysicalPresence(p))(state) IN
  IF member(LIFETIME_LOCK,p) OR member(HW_ENABLE,p) OR member(CMD_ENABLE,p)
     OR member(HW_DISABLE,p) OR member(CMD_DISABLE,p)
  THEN IF state`permFlags`physicalPresenceLifetimeLock
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(LOCK,p) OR member(PRESENT,p) OR member(NOTPRESENT,p)
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(HW_ENABLE,p) AND member(HW_DISABLE,p)
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(CMD_ENABLE,p) AND member(CMD_DISABLE,p)
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSE s=state WITH [`permFlags`physicalPresenceHWEnable:=
                                IF member(HW_ENABLE,p) THEN TRUE
                                ELSIF member(HW_DISABLE,p) THEN FALSE
                                ELSE s`permFlags`physicalPresenceHWEnable ENDIF,
                          `permFlags`physicalPresenceCMDEnable:=
                                IF member(CMD_ENABLE,p) THEN TRUE
                                ELSIF member(HW_DISABLE,p) THEN FALSE
                                ELSE s`permFlags`physicalPresenceCMDEnable ENDIF,
                          `permFlags`physicalPresenceLifetimeLock:=
                                IF member(LIFETIME_LOCK,p) THEN TRUE
                                ELSE s`permFlags`physicalPresenceLifetimeLock ENDIF] AND
            a=OUT_PhysicalPresence(TPM_SUCCESS)
       ENDIF
  ELSIF member(LOCK,p) OR member(PRESENT,p) OR member(NOTPRESENT,p)
  THEN IF member(LIFETIME_LOCK,p) OR member(HW_ENABLE,p) OR
          member(CMD_ENABLE,p)OR member(HW_DISABLE,p) OR member(CMD_DISABLE,p)
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF state`permFlags`physicalPresenceCMDEnable=FALSE
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(LOCK,p) AND member(PRESENT,p)
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(PRESENT,p) AND member(NOTPRESENT,p)
       THEN s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF state`stclearFlags`physicalPresenceLock
       THEN  s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
       ELSIF member(LOCK,p)
       THEN s=state WITH [`stclearFlags`physicalPresence:=FALSE
                         ,`stclearFlags`physicalPresenceLock:=TRUE] AND
            a=OUT_PhysicalPresence(TPM_SUCCESS)
       ELSIF member(PRESENT,p)
       THEN s=state WITH [`stclearFlags`physicalPresence:=TRUE] AND
            a=OUT_PhysicalPresence(TPM_SUCCESS)
       ELSIF member(NOTPRESENT,p)
       THEN s=state WITH [`stclearFlags`physicalPresence:=FALSE] AND
            a=OUT_PhysicalPresence(TPM_SUCCESS)
       ELSE s=state AND a=OUT_Error(TPM_BAD_PARAMETER) % Should be unreachable
```

```
      ENDIF
  ELSE s=state AND a=OUT_Error(TPM_BAD_PARAMETER)
  ENDIF;

TSC_ResetEstablishmentBit : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState): executeCom(s,ABS_ResetEstablishmentBit)),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_ResetEstablishmentBit)));

reset_establishment_post: THEOREM
  FORALL (state:(afterStartup?)) :
    LET (a,s) = runState(TSC_ResetEstablishmentBit)(state) IN
  IF resetEstablishment?(state)
  THEN a=OUT_ResetEstablishmentBit(TPM_SUCCESS) AND
       s=state WITH [`permFlags(tpmEstablished):=FALSE]
  ELSE a=OUT_Error(TPM_BAD_LOCALITY) AND s=state
  ENDIF;


%% Seal and output a blob (should be binary or data)
TPM_Seal(k:(tpmKey?),pcrInfo:(tpmPCRInfoLong?),inData:tpmData) : State =
   output(LAMBDA (s:tpmAbsState): outputCom(s,ABS_Seal(k,pcrInfo,inData)));

%% Seal a secret successfully
seal_post: THEOREM
  FORALL (state:(afterStartup?),k:(tpmKey?),p:(tpmPCRInfoLong?),d:tpmData) :
    LET (a,s) = runState(TPM_Seal(k,p,d))(state) IN
    IF not(storage?(keyUsage(k)) AND not(migratable(keyFlags(k))))
    THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
    ELSE LET sealInfo=tpmPCRInfoLong(state`stanyFlags`localityModifier,
                          locAtRelease(p),creationPCRSelect(p),releasePCRSelect(p),
                          tpmCompositeHash((#select:=creationPCRSelect(p),pcrValue:=state`pcrs#
                          digAtRelease(p)),
             a1=tempAuthData IN
         a=OUT_Seal(tpmStoredData(sealInfo,
                          tpmSealedData(a1,state`permData`tpmProof,
                               tpmDigest(cons(tpmStoredData(sealInfo,tpmSealedNull,clear),
                                              null),clear),
                               d,encrypted(key(k),clear)),
                          clear),
                    TPM_SUCCESS)
    ENDIF
    AND s=state;


%% Unseal and output a blob (should be binary or data)
TPM_Unseal(p:(tpmKey?),d:(tpmStoredData?)) : State =
   output(LAMBDA (s:tpmAbsState): outputCom(s,ABS_Unseal(p,d)));

unseal_post: THEOREM
```

```
    FORALL (state:(afterStartup?),p:(tpmKey?),d:(tpmStoredData?)) :
      LET (a,s) = runState(TPM_Unseal(p,d))(state) IN
      LET S2=tpmStoredData(sealInfo(d),tpmSealedNull,clear),
          H2=tpmCompositeHash((#select:=releasePCRSelect(sealInfo(d)),
                               pcrValue:=s`pcrs#)) IN
      IF not(storage?(keyUsage(p)) AND not(migratable(keyFlags(p))))
      THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
      ELSIF not(clear?(crs(decrypt(key(p),d))))
      THEN a=OUT_CPUError(CPU_DECRYPT_ERROR)
      ELSIF not(tpmProof(encData(d))=s`permData`tpmProof AND
            storedDigest(encData(d))=tpmDigest(cons(S2,null),clear))
      THEN a=OUT_Error(TPM_NOTSEALED_BLOB)
      ELSIF not(locAtRelease(sealInfo(S2))=s`stanyFlags`localityModifier)
      THEN a=OUT_Error(TPM_BAD_LOCALITY)
      ELSIF not(H2=digAtRelease(sealInfo(S2)))
      THEN a=OUT_Error(TPM_WRONGPCRVAL)
      ELSE a=OUT_Unseal(data(encData(d)),TPM_SUCCESS)
      ENDIF
      AND s=state;

  unseal_prev_post: THEOREM
    FORALL (state:(afterStartup?),j,k:(tpmKey?),p:(tpmPCRInfoLong?),d:tpmData) :
      LET (a,s) = runState(
                    TPM_Seal(k,p,d)
                    >>= LAMBDA (a:tpmAbsOutput) :
                           CASES a OF
                             OUT_Seal(sd,m) : TPM_Unseal(j,sd)
                             ELSE TPM_Noop(a)
                           ENDCASES)
                    (state) IN
seal?(k) AND unseal?(j) AND key(j)=private(k) AND
locAtRelease(p)=localityModifier(stanyFlags(state)) AND
digAtRelease(p)=tpmCompositeHash((#select:=releasePCRSelect(p),
                                      pcrValue:=state`pcrs#))
      =>
    a=OUT_Unseal(d,TPM_SUCCESS) AND
    s=state;

  %% UnBind encrypted blob (decrypt)
  TPM_UnBind(keyHandle:(tpmKey?),inData:(tpmBoundData?)) : State =
     output(LAMBDA (s:tpmAbsState):outputCom(s,ABS_UnBind(keyHandle,inData)));

  %% UnBind an encrypted blob successfully
  unBind_post: THEOREM
    FORALL (state:(afterStartup?),k:(tpmKey?),d:(tpmBoundData?)) :
      LET (a,s) = runState(TPM_UnBind(k,d))(state) IN
      LET d1=decrypt(private(k),d) IN
      IF not(legacy?(keyUsage(k)) OR bind?(keyUsage(k)))
      THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
      ELSIF not(clear?(crs(d1)))
```

```
    THEN a=OUT_CPUError(CPU_DECRYPT_ERROR)
    ELSE a=OUT_UnBind(payloadData(d),TPM_SUCCESS)
    ENDIF
    AND s=state;

Tspi_Data_Bind(k:(tpmKey?),d:tpmData) : State =
   output(LAMBDA (s:tpmAbsState): outputCom(s,ABS_Data_Bind(k,d)));

%% UnBind an encrypted blob successfully
unBind_prev_post: THEOREM
  FORALL (state:(afterStartup?),d:tpmData,p,k:(tpmKey?)) :
    LET (a,s) = runState(
                  Tspi_Data_Bind(k,d)
                  >>= LAMBDA (a:tpmAbsOutput) :
                        CASES a OF
                          OUT_Data_Bind(d,m) : TPM_UnBind(k,d)
                          ELSE TPM_Noop(a)
                        ENDCASES)
                  (state) IN
tpmBoundData?(d) AND unBind?(k) =>
a=OUT_UnBind(d,TPM_SUCCESS) AND
s=state;

%% Wrap new key k with parent key
TPM_CreateWrapKey(p,k:(tpmKey?)) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_CreateWrapKey(p,k))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_CreateWrapKey(p,k))));

create_wrap_key_post: THEOREM
  FORALL (state:(afterStartup?),p,k:(tpmKey?)) :
    LET (a,s) = runState(TPM_CreateWrapKey(p,k))(state) IN
  IF not storage?(keyUsage(p))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF migratable(keyFlags(p)) AND not(migratable(keyFlags(k)))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF identity?(keyUsage(k)) or authChange?(keyUsage(k))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF migrateAuthority(keyFlags(k))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF state`permFlags`FIPS AND
        (never?(authDataUsage(k)) OR legacy?(keyUsage(k)))
  THEN a=OUT_Error(TPM_NOTFIPS) AND s=state
  ELSIF (storage?(keyUsage(k)) OR migrate?(keyUsage(k))) AND
        not(RSA?(algoId(algoParms(k))))
  THEN a=OUT_Error(TPM_BAD_KEY_PROPERTY) AND s=state
  ELSE LET DU1=tempAuthData,
           DM1=tempAuthData,
           migAuth:(tpmSecret?)=IF migratable(keyFlags(k))
                                THEN DM1 ELSE state`permData`tpmProof ENDIF,
```

```
          h=tpmCompositeHash((#select:=creationPCRSelect(PCRInfo(k)),
                              pcrValue:=state`pcrs#)) IN
      LET encData=tpmStoreAsymkey(DU1,migAuth,pubDataDigest(encDat(k)),
                                  privKey(encDat(k)),clear),
          pcrs=tpmPCRInfoLong(state`locality,locAtRelease(PCRInfo(k)),
                              creationPCRSelect(PCRInfo(k)),
                              releasePCRSelect(PCRInfo(k)),
                              h,digAtRelease(PCRInfo(k))) IN
      a=OUT_CreateWrapKey(tpmKey(state`keyGenCnt,keyUsage(k),keyFlags(k),
                                 authDataUsage(k),algoParms(k),pcrs,key(p),
                                 encData,clear),
                          TPM_SUCCESS) AND
      s=state WITH [`keyGenCnt:=keyGenCnt(state)+1]
  ENDIF;

%% Install key k in a TPM
TPM_LoadKey2(p,k:(tpmKey?)):State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState):executeCom(s,ABS_LoadKey2(p,k))),
    (LAMBDA (s:tpmAbsState):outputCom(s,ABS_LoadKey2(p,k))));

% A key is installed if it is wrapped with SRK
load_key_post: THEOREM
  FORALL (state:(afterStartup?),p,k:(tpmKey?)) :
    LET (a,s) = runState(TPM_LoadKey2(p,k))(state) IN
  IF not storage?(keyUsage(p))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF wrappingKey(k)/=key(p)
  THEN a=OUT_CPUError(CPU_DECRYPT_ERROR) AND s=state
  ELSE CASES keyUsage(k) OF
         identity: IF migratable(keyFlags(k))=FALSE
                   THEN IF migratable(keyFlags(k))=0 AND
                           migrationAuth(encDat(k))/=tpmProof(permData(state))
                        THEN a=OUT_Error(TPM_FAIL) AND s=state
                        ELSIF FIPS(permFlags(state)) AND
                              (never?(authDataUsage(k)) OR legacy?(keyUsage(k)))
                        THEN a=OUT_Error(TPM_NOTFIPS) AND s=state
                        ELSE a=OUT_LoadKey2(k,TPM_SUCCESS) AND
                             s=state WITH [`keys:=loadKey(k,p,state`keys,state`pcrs)]
                        ENDIF
                   ELSE a=OUT_Error(TPM_INVALID_KEYUSAGE) AND
                        s=state
                   ENDIF,
         authChange: a=OUT_Error(TPM_INVALID_KEYUSAGE) AND
                       s=state
         ELSE IF migratable(keyFlags(k))=0 AND
                 migrationAuth(encDat(k))/=tpmProof(permData(state))
              THEN a=OUT_Error(TPM_FAIL) AND s=state
              ELSIF FIPS(permFlags(state)) AND (never?(authDataUsage(k)) OR
                    legacy?(keyUsage(k)))
```

```
              THEN a=OUT_Error(TPM_NOTFIPS) AND s=state
              ELSE a=OUT_LoadKey2(k,TPM_SUCCESS) AND
                   s=state WITH [`keys:=loadKey(k,p,state`keys,state`pcrs)]
              ENDIF
         ENDCASES
  ENDIF;

load_key_pred_test: THEOREM
  FORALL (state:(afterStartup?),p,k:(tpmKey?)) :
    LET (a,s) = runState(TPM_LoadKey2(p,k))(state) IN
  IF loadKey2?(state,p,k)
  THEN a=OUT_LoadKey2(k,TPM_SUCCESS) AND
       IF key(p)=key(TPM_KH_SRK)
       THEN s=state WITH [`keys:=add(key(k),state`keys)]
       ELSIF member(key(p),state`keys)
       THEN s=state WITH [`keys:=add(key(k),state`keys)]
       ELSE s=state
       ENDIF
  ELSE IF not storage?(keyUsage(p))
       THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
       ELSIF key(p)/=wrappingKey(k)
       THEN a=OUT_CPUError(CPU_DECRYPT_ERROR)
       ELSIF identity?(keyUsage(k)) AND migratable(keyFlags(k))
       THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
       ELSIF authChange?(keyUsage(k))
       THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
       ELSIF not(migratable(keyFlags(k))) AND
             migrationAuth(encDat(k))/=tpmProof(permData(state))
       THEN a=OUT_Error(TPM_FAIL)
       ELSIF FIPS(permFlags(state)) AND (never?(authDataUsage(k)) OR
             legacy?(keyUsage(k)))
       THEN a=OUT_Error(TPM_NOTFIPS)
       ELSE a=OUT_Error(TPM_SUCCESS)
       ENDIF
       AND s=state
  ENDIF;

load_key_post2: THEOREM
  FORALL (state:(afterStartup?),p,k:(tpmKey?)) :
    LET (a,s) = runState(TPM_LoadKey2(p,k))(state) IN
       not (loadKey2?(state,p,k)) =>
         not (a=OUT_Error(TPM_SUCCESS));

%% A key is installed if it is wrapped with an installed key
%% proved - Tue Jun 12 15:38:22 CDT 2012
load_key_post3: THEOREM
  FORALL (state:(afterStartup?),p,j,k:(tpmKey?)) :
    LET (a,s) = runState(
                  TPM_LoadKey2(srk(state),k)
                  >> TPM_LoadKey2(k,j))
```

```
                  (state) IN
  loadKey2?(state,srk(state),k) AND loadKey2?(state,k,j) AND srk(state)=TPM_KH_SRK
      => member(key(j),s`keys);

create_load_key_post: THEOREM
  FORALL (state:(afterStartup?),k,p:(tpmKey?),x:nat,d:PCRVALUES) :
    LET (a,s) = runState(
                  TPM_CreateWrapKey(p,k)
                  >>= CPU_saveOutput(x)
                  >>= LAMBDA (a:tpmAbsOutput) :
                        CASES a OF
                          OUT_CreateWrapKey(wk,m) : TPM_LoadKey2(p,wk)
                          ELSE TPM_Noop(a)
                        ENDCASES)
                  (state) IN
    OUT_CreateWrapKey?(s`memory(x)) AND
    createWrapKey?(p,k) AND loadKey2?(state,p,wrappedKey(s`memory(x))) =>
    a=OUT_LoadKey2(wrappedKey(s`memory(x)),TPM_SUCCESS) AND
    s=state WITH[`keyGenCnt:=state`keyGenCnt+1
                ,`keys:=IF member(key(p),state`keys) OR key(p)=key(TPM_KH_SRK)
                        THEN addKey(wrappedKey(s`memory(x)),state`keys)
                        ELSE state`keys ENDIF
                ,`memory:=s`memory];

%% Install key k in a TPM
TPM_GetPubKey(k:(tpmKey?)):State =
  output((LAMBDA (s:tpmAbsState):outputCom(s,ABS_GetPubKey(k))));

get_pub_key_post: THEOREM
 FORALL (state:(afterStartup?),k:(tpmKey?)) :
  LET (a,s) = runState(TPM_GetPubKey(k))(state) IN
  IF getPubKey?(state,k)
  THEN a=OUT_GetPubKey(tpmPubkey(key(k)),TPM_SUCCESS)
  ELSIF not never?(authDataUsage(k))
  THEN a=OUT_Error(TPM_AUTHFAIL)
  ELSIF s`permFlags`readSRKPub=FALSE
  THEN a=OUT_Error(TPM_INVALID_KEYHANDLE)
  ELSIF pcrIgnoredOnRead(keyFlags(k))=FALSE AND
        dig(digAtRelease(PCRInfo(k)))/=
            dig(digAtRelease(PCRInfo(k))) WITH [pcrValue:=s`pcrs]
  THEN a=OUT_Error(TPM_WRONGPCRVAL)
  ELSE a=OUT_Error(TPM_SUCCESS)
  ENDIF
  AND s=state;

get_pub_key_post2: THEOREM
 FORALL (state:(afterStartup?),k:(tpmKey?)) :
  LET (a,s) = runState(TPM_GetPubKey(k))(state) IN
  not getPubKey?(state,k) =>
    not (a=OUT_Error(TPM_SUCCESS));
```

```
TPM_CreateMigrationBlob(pk:(tpmKey?),m:migrateScheme,mka:(tpmMigKeyAuth?),
  d:(tpmKey?)) : State =
   output(LAMBDA (s:tpmAbsState) :
    outputCom(s,ABS_CreateMigrationBlob(pk,m,mka,d)));

create_mig_blob_post: THEOREM
 FORALL (state:(afterStartup?),pk:(tpmKey?),m:migrateScheme,
    mka:(tpmMigKeyAuth?),d:(tpmKey?)) :
  LET(a,s) = runState(TPM_CreateMigrationBlob(pk,m,mka,d))(state) IN
  IF createMigBlob?(state,pk,m,mka,d)
  THEN LET d1:(tpmKey?)=decrypt(key(pk),d) IN
       CASES m OF
         migrate : a=OUT_CreateMigrationBlob(rand,
                         tpmMigrateAsymkey(usageAuth(encDat(d1)),
                                           pubDataDigest(encDat(d1)),
                                           privKey(encDat(d1))),
                         TPM_SUCCESS),
         rewrap  : a=OUT_CreateMigrationBlob(0,
                         CASES d OF
                           tpmKey(k,u,f,a,ap,p,w,e,c) :
                               tpmKey(k,u,f,a,ap,p,key(key(mka)),e,c)
                         ENDCASES,
                         TPM_SUCCESS)
       ENDCASES
  ELSIF not storage?(keyUsage(pk))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
  ELSIF not checkMigKeyAuth?(s,mka)
  THEN a=OUT_Error(TPM_AUTHFAIL)
  ELSE a=OUT_Error(TPM_BAD_PARAMETER)
  ENDIF
  AND s=state;

TPM_ConvertMigrationBlob(p:(tpmKey?),inData:(tpmMigrateAsymkey?),r:int):State=
   output(LAMBDA (s:tpmAbsState) :
    outputCom(s,ABS_ConvertMigrationBlob(p,inData,r)));

convert_mig_blob_post: THEOREM
 FORALL (state:(afterStartup?),p:(tpmKey?),d:(tpmMigrateAsymkey?),r:int) :
  LET (a,s) = runState(TPM_ConvertMigrationBlob(p,d,r))(state) IN
  IF storage?(keyUsage(p))
  THEN LET d1=decrypt(key(p),d),
           pHash=tempAuthData,
           k1=partPrivKey(d) IN
       LET d2=tpmStoreAsymkey(usageAuth(d),pHash,pubDataDigest(d),k1,
                        encrypted(key(p),clear)) IN
       a=OUT_ConvertMigrationBlob(d2,TPM_SUCCESS)
  ELSE a=OUT_Error(TPM_INVALID_KEYUSAGE)
  ENDIF
  AND s=state;
```

```
TPM_AuthorizeMigrationKey(migKey:(tpmKey?),migScheme:(tpmMigScheme?)) : State=
  output(LAMBDA (s:tpmAbsState) :
    outputCom(s,ABS_AuthorizeMigrationKey(migKey,migScheme)));

authorize_migration_key_post: THEOREM
 FORALL (state:(afterStartup?),k:(tpmKey?),scheme:(tpmMigScheme?)):
  LET(a,s) = runState(TPM_AuthorizeMigrationKey(k,scheme))(state) IN
  a=OUT_AuthorizeMigrationKey(
           tpmMigKeyAuth(k,scheme,
                tpmDigest(cons(k,cons(scheme,cons(s`permData`tpmProof,
                    null))),clear),
                clear),
           TPM_SUCCESS)
  AND s=state;

TPM_MigrateKey(migKey,pubKey:(tpmKey?),data:tpmData) : State =
  output(LAMBDA (s:tpmAbsState) :
    outputCom(s,ABS_MigrateKey(migKey,pubKey,data)));

migrate_key_post: THEOREM
 FORALL (state:(afterStartup?),mk,k:(tpmKey?),d:tpmData) :
  LET(a,s) = runState(TPM_MigrateKey(mk,k,d))(state) IN
  IF migrate?(keyUsage(mk))
  THEN a = OUT_MigrateKey(encrypt(key(k),decrypt(key(mk),d)),TPM_SUCCESS)
  ELSE a = OUT_Error(TPM_INVALID_KEYUSAGE)
  ENDIF
  AND s=state;

%% Generate and output a signature
TPM_Sign(k:(tpmKey?),areaToSign:tpmData) : State =
  output(LAMBDA (s:tpmAbsState) : outputCom(s,ABS_Sign(k,areaToSign)));

sign_post: THEOREM
  FORALL (state:(afterStartup?),k:(tpmKey?),b:tpmData) :
    LET(a,s) = runState(TPM_Sign(k,b))(state) IN
    IF legacy?(keyUsage(k)) or signing?(keyUsage(k))
    THEN a = OUT_Sign(sign(key(k),b),TPM_SUCCESS)
    ELSE a = OUT_Error(TPM_INVALID_KEYUSAGE)
    ENDIF
    AND s = state;

sign_pred_test : THEOREM
  FORALL (state:(afterStartup?),k:(tpmKey?),b:tpmData) :
    LET(a,s) = runState(TPM_Sign(k,b))(state) IN
    IF sign?(state,k,b)
    THEN a = OUT_Sign(sign(key(k),b),TPM_SUCCESS)
    ELSE a = OUT_Error(TPM_INVALID_KEYUSAGE)
    ENDIF
    AND s = state;
```

```
TPM_CreateEndorsementKeyPair(n:(tpmNonce?),k:(tpmKey?)) : State =
  modifyOutput(
    (LAMBDA(s:tpmAbsState):executeCom(s,ABS_CreateEndorsementKeyPair(n,k))),
    (LAMBDA(s:tpmAbsState):outputCom(s,ABS_CreateEndorsementKeyPair(n,k))));

create_endorsement_key_pair_post: THEOREM
 FORALL (state:(afterStartup?),k:(tpmKey?),n:(tpmNonce?)) :
  LET (a,s) = runState(TPM_CreateEndorsementKeyPair(n,k))(state) IN
  IF key(ek(state))=badkey
  THEN a=OUT_CreateEndorsementKeyPair(k,
                    tpmDigest(cons(k,cons(n,null)),clear),TPM_SUCCESS) AND
       s=state WITH [‘ek := privateKey(k)
                    ,‘permFlags(CEKPUsed) := TRUE
                    ,‘permFlags(enableRevokeEK) := FALSE]
  ELSE a=OUT_Error(TPM_DISABLED_CMD) AND
       s=state
  ENDIF;


TPM_CreateRevocableEK(a:(tpmNonce?),k:(tpmKey?),g:bool,i:(tpmNonce?)) :State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_CreateRevocableEK(a,k,g,i))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_CreateRevocableEK(a,k,g,i))));

create_revocable_ek_post: THEOREM
 FORALL (state:(afterStartup?),a,i:(tpmNonce?),k:(tpmKey?),g:bool) :
  LET (a,s) = runState(TPM_CreateRevocableEK(a,k,g,i))(state) IN
  IF key(ek(state)) = badkey
  THEN a=OUT_CreateRevocableEK(tpmPubkey(key(k)),
                               tpmDigest(cons(k,cons(a,null)),clear),
                               s‘permData‘ekReset,TPM_SUCCESS) AND
       s=state WITH [‘ek := privateKey(k)
                    ,‘permFlags(CEKPUsed) := TRUE
                    ,‘permFlags(enableRevokeEK) := TRUE
                    ,‘permData(ekReset):=IF g THEN tpmNonce(rand)
                                          ELSE i ENDIF]
  ELSE a=OUT_Error(TPM_DISABLED_CMD) AND
       s=state
  ENDIF;

TPM_RevokeTrust(r:(tpmNonce?)) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState) : executeCom(s,ABS_RevokeTrust(r))),
    (LAMBDA (s:tpmAbsState) : outputCom(s,ABS_RevokeTrust(r))));

revoke_trust_post : THEOREM
 FORALL (state:(afterStartup?),EKReset:(tpmNonce?)) :
  LET (a,s) = runState(TPM_RevokeTrust(EKReset))(state) IN
  IF revokeTrust?(state,EKReset)
```

```
  THEN a=OUT_RevokeTrust(TPM_SUCCESS) AND
   LET s1=clear(state) IN
   s=s1 WITH [‘permFlags(nvLocked):=FALSE
                ,‘ek:=tpmKey(0,keyUsage(s‘ek),keyFlags(s‘ek),
                             authDataUsage(s‘ek),algoParms(s‘ek),
                             PCRInfo(s‘ek),wrappingKey(s‘ek),
                             encDat(s‘ek),crs(s‘ek))]
  ELSIF not state‘permFlags‘enableRevokeEK
  THEN a=OUT_Error(TPM_PERMANENTEK) AND s=state
  ELSIF state‘permData‘ekReset/=EKReset
  THEN a=OUT_Error(TPM_AUTHFAIL) AND s=state
  ELSIF not s‘stclearFlags‘physicalPresence
  THEN a=OUT_Error(TPM_BAD_MODE) AND s=state
  ELSE a=OUT_Error(TPM_SUCCESS) AND s=state
  ENDIF;

revoke_trust_post2 : THEOREM
 FORALL (state:(afterStartup?),EKReset:(tpmNonce?)) :
  LET (a,s) = runState(TPM_RevokeTrust(EKReset))(state) IN
  not revokeTrust?(state,EKReset) =>
    not a=OUT_Error(TPM_SUCCESS);

TPM_ReadPubek(n:(tpmNonce?)) : State =
   output(LAMBDA (s:tpmAbsState) : outputCom(s,ABS_ReadPubek(n)));

read_pub_ek_post: THEOREM
  FORALL (state:(afterStartup?),n:(tpmNonce?)) :
    LET (a,s) = runState(TPM_ReadPubek(n))(state) IN
    a=IF readPubek?(state,n)
      THEN LET pubEndoK=ek(state) IN
           OUT_ReadPubek(pubEndoK,tpmDigest(cons(pubEndoK,cons(n,null)),clear),
                  TPM_SUCCESS)
      ELSIF not state‘permFlags‘readPubek
      THEN OUT_Error(TPM_DISABLED_CMD)
      ELSIF not goodkey?(key(ek(state)))
      THEN OUT_Error(TPM_NO_ENDORSEMENT)
      ELSE OUT_Error(TPM_SUCCESS)
      ENDIF
    AND s=state;

read_pub_ek_post2: THEOREM
  FORALL (state:(afterStartup?),n:(tpmNonce?)) :
    LET (a,s) = runState(TPM_ReadPubek(n))(state) IN
  not readPubek?(state,n) =>
    not a=OUT_Error(TPM_SUCCESS);

read_pub_ek_take_ownership: THEOREM
  FORALL (state:(afterStartup?),n:(tpmNonce?),srk:(tpmKey?)) :
    LET (a,s) = runState(
                  TPM_TakeOwnership(srk)
```

```
                >> TPM_ReadPubek(n))
                (state) IN
  takeOwnership?(state,srk) =>
    a=OUT_Error(TPM_DISABLED_CMD);

TPM_OwnerReadInternalPub(k:(tpmKey?)) : State =
  output(LAMBDA (s:tpmAbsState) : outputCom(s,ABS_OwnerReadInternalPub(k)));

owner_read_internal_pub_post: THEOREM
  FORALL (state:(afterStartup?),k:(tpmKey?)) :
    LET (a,s) = runState(TPM_OwnerReadInternalPub(k))(state) IN
    IF key(k)=ekKeyVal
    THEN a=OUT_OwnerReadInternalPub(tpmPubkey(ekKeyVal),TPM_SUCCESS)
    ELSIF key(k)=srkKeyVal
    THEN a=OUT_OwnerReadInternalPub(tpmPubkey(srkKeyVal),TPM_SUCCESS)
    ELSE a=OUT_Error(TPM_BAD_PARAMETER)
    ENDIF
    AND s=state;

%% Make a new identity and output it
TPM_MakeIdentity(d:(tpmDigest?),k:(tpmKey?)) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState):executeCom(s,ABS_MakeIdentity(d,k))),
    (LAMBDA (s:tpmAbsState):outputCom(s,ABS_MakeIdentity(d,k))));

make_identity_post: THEOREM
 FORALL (state:(afterStartup?),d:(tpmDigest?),k:(tpmKey?)) :
  LET (a,s) = runState(TPM_MakeIdentity(d,k))(state) IN
  IF makeIdentity?(state,k)
  THEN LET A1=tempAuthData,
           pcr=tpmPCRInfoLong(makeIdentityLocality,locAtRelease(PCRInfo(k)),
                       creationPCRSelect(PCRInfo(k)),releasePCRSelect(PCRInfo(k)),
                       tpmCompositeHash((#select:=creationPCRSelect(PCRInfo(k)),
                                         pcrValue:=pcrs(state)#)),
                       digAtRelease(PCRInfo(k))),
           encData=tpmStoreAsymkey(A1,s`permData`tpmProof,pubDataDigest(encDat(k)),
                                   privKey(encDat(k)),clear) IN
       LET waik=tpmKey(keyGenCnt(state),keyUsage(k),keyFlags(k),authDataUsage(k),
                       algoParms(k),pcr,wrappingKey(srk(state)),encData,clear) IN
    LET idBind=tpmIdContents(d,waik,signed(private(waik),clear)) IN
        a=OUT_MakeIdentity(waik,idBind,TPM_SUCCESS) AND
        s=state WITH [`keyGenCnt := keyGenCnt(state)+1]
  ELSIF state`permFlags`FIPS AND never?(authDataUsage(k))
  THEN a=OUT_Error(TPM_NOTFIPS) AND s=state
  ELSIF not identity?(keyUsage(k))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSIF migratable(keyFlags(k))
  THEN a=OUT_Error(TPM_INVALID_KEYUSAGE) AND s=state
  ELSE a=OUT_Error(TPM_SUCCESS) AND s=state
  ENDIF;
```

```
make_identity_post2 : THEOREM
 FORALL (state:(afterStartup?),d:(tpmDigest?),k:(tpmKey?)) :
  LET (a,s) = runState(TPM_MakeIdentity(d,k))(state) IN
  not makeIdentity?(state,k) =>
   not a=OUT_Error(TPM_SUCCESS)

make_cert : THEOREM
 FORALL (state:(afterStartup?),d:(tpmDigest?),idk:(tpmKey?),
         id:(tpmIdContents?),x:nat) :
  LET (a,s) = runState(
                 TPM_MakeIdentity(d,idk)
                 >>= CPU_saveOutput(x)
                 >>= LAMBDA (a:tpmAbsOutput) :
                       CASES a OF
                         OUT_MakeIdentity(k,i,m) : CA_certify(k,i)
                         ELSE TPM_Noop(a)
                       ENDCASES)
                 (state) IN
  makeIdentity?(state,idk) AND
  OUT_MakeIdentity?(s`memory(x)) AND
  certify?(idKey(s`memory(x)),idBinding(s`memory(x)))
    =>
   a=OUT_Certify(idKey(s`memory(x)),
                 tpmAsymCAContents(tpmSessKey(keyGenCnt(state)+1,clear),
                                   digest(idBinding(s`memory(x))),
                                   encrypted(key(ekVal),clear)),
                 CPU_SUCCESS)
   AND s=state WITH [`keyGenCnt:=state`keyGenCnt+2
                    ,`memory:=s`memory];

%% Use an AIK
TPM_ActivateIdentity(a:(tpmKey?),b:(activateIdentityBlob?)) : State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState): executeCom(s,ABS_ActivateIdentity(a,b))),
    (LAMBDA (s:tpmAbsState): outputCom(s,ABS_ActivateIdentity(a,b))));

%% Assure that activate idenity installs its AIK and returns K if AIK can
%% be installed.
activate_identity_post: THEOREM
 FORALL (state:(afterStartup?),aik:(tpmKey?),blob:(activateIdentityBlob?)) :
  LET (a,s) = runState(TPM_ActivateIdentity(aik,blob))(state) IN
  IF activateIdentity?(state,aik,blob)
  THEN a=CASES blob OF
           tpmAsymCAContents(k,d,c) : OUT_ActivateIdentity(k,TPM_SUCCESS),
           tpmEKBlob(b,c) : OUT_ActivateIdentity(sessK(b),TPM_SUCCESS)
         ENDCASES
  ELSIF not identity?(keyUsage(aik))
  THEN a=OUT_Error(TPM_BAD_PARAMETER)
  ELSE LET h1=tpmDigest(cons(tpmPubkey(key(aik)),null),clear),
```

```
          b1=decrypt(private(ekVal),blob) IN
      CASES b1 OF
        tpmAsymCAContents(k,d,crs) : a=OUT_CPUError(CPU_DECRYPT_ERROR),
        tpmEKBlob(b,crs) :
            CASES b OF
              tpmEKBlobActivate(k,d,p) :
                    LET C1=tpmCompositeHash((#select:=pcrSelect(p),
                                             pcrValue:=pcrs(state)#)) IN
                    IF h1/=d
                    THEN a=OUT_Error(TPM_BAD_PARAMETER)
                    ELSIF not null?(pcrSelect(p)) AND C1/=digAtRelease(p)
                    THEN a=OUT_Error(TPM_WRONGPCRVAL)
                    ELSE a=OUT_Error(TPM_BAD_LOCALITY)
                    ENDIF
              ELSE a=OUT_Error(TPM_BAD_TYPE)
            ENDCASES
      ELSE a=OUT_Error(TPM_BAD_PARAMETER)
      ENDCASES
  ENDIF
  AND s=state;

activate_identity_post2 : THEOREM
 FORALL (state:(afterStartup?),aik:(tpmKey?),blob:(activateIdentityBlob?)) :
  LET (a,s) = runState(TPM_ActivateIdentity(aik,blob))(state) IN
  not activateIdentity?(state,aik,blob) =>
   not a=OUT_Error(TPM_SUCCESS)

cert_activate : THEOREM
  FORALL(state:(afterStartup?),aik:(tpmKey?),c:(tpmIdContents?),x:nat):
    LET (a,s) = runState(
                  CA_certify(aik,c)
                  >>= CPU_saveOutput(x)
                  >>= LAMBDA (a:tpmAbsOutput) :
                        CASES a OF
                          OUT_Certify(k,d,m) : TPM_ActivateIdentity(k,d)
                          ELSE TPM_Noop(a)
                        ENDCASES)
                  (state) IN
  certify?(aik,c) AND
  OUT_Certify?(s`memory(x)) AND
  activateIdentity?(s,k(s`memory(x)),dat(s`memory(x)))
  =>
  a=OUT_ActivateIdentity(sessK(dat(s`memory(x))),TPM_SUCCESS) AND
  s=state WITH [`keyGenCnt:=1+keyGenCnt(state)
               ,`memory:=updateLoc(state`memory,x,
                           OUT_Certify(aik,tpmAsymCAContents(
                                           tpmSessKey(keyGenCnt(state),clear),
                                                      digest(c),
                                                      encrypted(key(ekVal),clear)),
                                       CPU_SUCCESS))];
```

```
make_cert_activate_identity: THEOREM
 FORALL (state:(afterStartup?),d:(tpmDigest?),k:(tpmKey?),x,y:nat) :
  LET (a,s) = runState(
                  TPM_MakeIdentity(d,k)
                  >>= CPU_saveOutput(x)
                  >>= LAMBDA (a:tpmAbsOutput) :
                        CASES a OF
                          OUT_MakeIdentity(k,i,m) : CA_certify(k,i)
                          ELSE TPM_Noop(a)
                        ENDCASES
                  >>= CPU_saveOutput(y)
                  >>= LAMBDA (a:tpmAbsOutput) :
                        CASES a OF
                          OUT_Certify(j,d,m) : TPM_ActivateIdentity(j,d)
                          ELSE TPM_Noop(a)
                        ENDCASES)
                  (state) IN
  makeIdentity?(state,k) AND
  OUT_MakeIdentity?(s`memory(x)) AND
  certify?(idKey(s`memory(x)),idBinding(s`memory(x))) AND
  OUT_Certify?(s`memory(y)) AND
  wellFormedRestore?(s`restore) AND
  activateIdentity?(tpmRestore(s`restore),idKey(s`memory(x)),dat(s`memory(y)))
  =>
  a=OUT_ActivateIdentity(sessK(dat(s`memory(y))),TPM_SUCCESS) AND
  s=state WITH [`memory:=s`memory
               ,`keyGenCnt:=state`keyGenCnt+2]

%% Extend PCR n with hash value h.
TPM_Extend(n:PCRINDEX,h:HV):State =
  modifyOutput(
    (LAMBDA (s:tpmAbsState):executeCom(s,ABS_Extend(n,h))),
    (LAMBDA (s:tpmAbsState):outputCom(s,ABS_Extend(n,h))));

extend_post : THEOREM
 FORALL (state:(afterStartup?),h:HV,n:PCRINDEX) :
  LET (a,s) = runState(TPM_Extend(n,h))(state) IN
  LET L1=state`stanyFlags`localityModifier,
      P1=pcrExtendLocal(state`permData`pcrAttrib(n)),
      H1=pcrsExtend(state`pcrs,n,h) IN
  IF extend?(state,n,h)
  THEN IF state`permFlags`disable OR state`stclearFlags`deactivated
       THEN a=OUT_Extend(reset,TPM_SUCCESS) AND s=state WITH [`pcrs:=H1]
       ELSE a=OUT_Extend(extend(state`pcrs(n),h),TPM_SUCCESS) AND
            s=state WITH [`pcrs:=H1]
       ENDIF
  ELSIF n>23 OR n<0
  THEN a=OUT_Error(TPM_BADINDEX) AND
       s=state
```

```
  ELSIF not member(L1,P1)
  THEN a=OUT_Error(TPM_BAD_LOCALITY) AND
       s=state
  ELSE a=OUT_Error(TPM_SUCCESS) AND s=state
  ENDIF;

extend_post2 : THEOREM
 FORALL (state:(afterStartup?),h:HV,n:PCRINDEX) :
  LET (a,s) = runState(TPM_Extend(n,h))(state) IN
  LET L1=s`stanyFlags`localityModifier,
      P1=pcrExtendLocal(state`permData`pcrAttrib(n)),
      H1=pcrsExtend(state`pcrs,n,h) IN
  not extend?(state,n,h) =>
    not a=OUT_Error(TPM_SUCCESS)

%% Extend is antisymmetric if the hash values used for extension are
%% not equal.  In this example we use the LET form to define
%% commands directly with monad functions defined in StateMonad.pvs
antisymmetryMonad: THEOREM
 FORALL (state:(afterStartup?),hv0,hv1:HV,n:PCRINDEX) :
  LET f1:State = modify(OUT_Nothing,(LAMBDA (s:tpmAbsState):
                            s WITH [`pcrs:=pcrsExtend(pcrs(s),n,hv0)])),
      f2:State = modify(OUT_Nothing,(LAMBDA (s:tpmAbsState):
                            s WITH [`pcrs:=pcrsExtend(pcrs(s),n,hv1)])) IN
  hv0/=hv1 =>
    runState(f2 >>= (LAMBDA (x:tpmAbsOutput): f1))(tpmStartup)
    /=
    runState(f1 >>= (LAMBDA (x:tpmAbsOutput): f2))(tpmStartup);

%% Extending a reset PCR is antisymmetric if the two values are not equal
%% Let form used to defined commands for bind.  Will use command forms
%% later in antisymmetryMonad3
antisymmetryMonad2: THEOREM
 FORALL (state:(afterStartup?),hv0,hv1:HV,n:PCRINDEX) :
  LET f1:State=TPM_Extend(n,hv0),
      f2:State=TPM_Extend(n,hv1) IN
  extend?(state,n,hv0) AND extend?(state,n,hv1) AND
  not(state`stclearFlags`deactivated OR state`permFlags`disable) AND
  %IF flags are set, the result will be same (OUT_Extend(reset,TPM_SUCCESS))
  hv0/=hv1 =>
    runState(f2 >> f1)(state)
    /=
    runState(f1 >> f2)(state)

%% Extending a reset PCR is antisymmetric if the two values are not equal.
%% No let form - commands appear directly in bind
%% Theorem is now false when PCRs other than PCR0 are included.
antisymmetryMonad3: THEOREM
 FORALL (state:(afterStartup?),hv0,hv1:HV,n:PCRINDEX) :
  extend?(state,n,hv0) AND extend?(state,n,hv1) AND
```

```
  not(state`stclearFlags`deactivated OR state`permFlags`disable) AND
  %IF flags are set, the result will be same (OUT_Extend(reset,TPM_SUCCESS))
  hv0/=hv1 =>
     runState(TPM_Extend(n,hv1)
   >> TPM_Extend(n,hv0))
   (state)
   /=
     runState(TPM_Extend(n,hv0)
   >> TPM_Extend(n,hv1))
   (state)

%% Output PCR(i)
TPM_PcrRead(i:PCRINDEX) : State =
   output(LAMBDA (s:tpmAbsState) : outputCom(s,ABS_PCRRead(i)));

pcr_read_post: THEOREM
  FORALL (state:(afterStartup?),i:PCRINDEX) :
    LET(a,s) = runState(TPM_PcrRead(i))(state) IN
  IF i > 23 OR i < 0
  THEN a=OUT_Error(TPM_BADINDEX)
  ELSE a=OUT_PCRRead(state`pcrs(i),TPM_SUCCESS)
  ENDIF
  AND s=state;


%% Generate and output a quote
TPM_Quote(k:(tpmKey?),n:(tpmNonce?),pm:PCR_SELECTION) : State =
   output(LAMBDA (s:tpmAbsState) : outputCom(s,ABS_Quote(k,n,pm)));

%% If nonces don't match, quotes don't match.  We don't care about this
%% if the a quote cannot be produced.
%% proved - Fri Jun 22 16:55:16 CDT 2012
bad_nonce: THEOREM
  FORALL (s:(afterStartup?),k:(tpmKey?),n1,n2:(tpmNonce?),pm:PCR_SELECTION) :
    n1/=n2 AND quote?(k)
    =>
     runState(TPM_Quote(k,n1,pm))(s)
     /=
     runState(TPM_Quote(k,n2,pm))(s);


%% Bad Signing Key - Man in the Middle Attack - we don't care about this
%% if either key won't produce a quote.
%% proved - Fri Jun 22 16:55:56 CDT 2012
bad_signing_key: THEOREM
  FORALL (s:(afterStartup?),n:(tpmNonce?),pm:PCR_SELECTION,k0,k1:(tpmKey?)) :
    private(k0) /= private(k1)  AND
     quote?(k0) AND quote?(k1)
      =>
    runState(TPM_Quote(k0,n,pm))(s)
```

```
      /=
      runState(TPM_Quote(k1,n,pm))(s);

  %% Bad PCRs - If different pcrs selected, different quote is output
  bad_pcrs: THEOREM
    FORALL (s:(afterStartup?),n:(tpmNonce?),p0,p1:PCR_SELECTION,k:(tpmKey?)) :
      quote?(k) AND p0/=p1
      =>
      runState(TPM_Quote(k,n,p0))(s)
      /=
      runState(TPM_Quote(k,n,p1))(s);


  %% Output after going through tpm commands is same as grabbing pcrs
  %% proved - Fri Jun 22 16:56:08 CDT 2012
  %% TODO: problem: quote isn't getting correct pcrs input since that's saved in intermediate sta
  check_PCRS: THEOREM
    FORALL (state:(afterStartup?),k:(tpmKey?),hv:HV,pm:PCR_SELECTION,
n:(tpmNonce?),ind:PCRINDEX,x:nat) :
      LET (a,s)=runState(
                    TPM_Extend(ind,hv)
                    >> TPM_Quote(k,n,pm))
                    (state) IN
    extend?(state,ind,hv) AND quote?(k) =>
    a=OUT_Quote(map(s`pcrs,pm),
                tpmQuote(tpmCompositeHash((#select:=pm,pcrValue:=s`pcrs#)),
                         n,
                         signed(private(k),clear)),
                TPM_SUCCESS) AND
    s=state WITH [`pcrs:=pcrsExtend(state`pcrs,ind,hv)
           ,`memory:=s`memory];

  %% Prove that quote generation returns the correct PCR.  This theorem
  %% needs to be updated with something cleaner.
  %% proven - Wed Jul 11 10:40:28 EDT 2012
  quote_post: THEOREM
    FORALL (state:(afterStartup?),k:(tpmKey?),n:(tpmNonce?),p:PCR_SELECTION) :
        LET (a,s) = runState(TPM_Quote(k,n,p))(state) IN
    IF not quote?(k)
    THEN a=OUT_Error(TPM_INVALID_KEYUSAGE)
    ELSE LET H1=tpmCompositeHash((#select:=p,pcrValue:=s`pcrs#)),
                                    pcrData=getPCRs(s`pcrs,p) IN
             a=OUT_Quote(pcrData,
                         tpmQuote(H1,n,signed(private(k),clear)),
                         TPM_SUCCESS)
ENDIF
AND s=state;

  quote_with_prev_key: THEOREM
   FORALL (state:(afterStartup?),pk,k:(tpmKey?),n:(tpmNonce?),pm:PCR_SELECTION,x:nat) :
```

```
    LET (a,s) = runState(
                    TPM_CreateWrapKey(pk,k)
                    >>=CPU_saveOutput(x)
                    >>= LAMBDA (a:tpmAbsOutput) :
                          CASES a OF
                            OUT_CreateWrapKey(wk,m) : TPM_Quote(wk,n,pm)
                            ELSE TPM_Noop(a)
                          ENDCASES)
                    (state) IN
    OUT_CreateWrapKey?(s`memory(x)) =>
      LET key=wrappedKey(s`memory(x)) IN
    createWrapKey?(pk,k) AND quote?(key)
  =>
      a=OUT_Quote(map(s`pcrs,pm),
                  tpmQuote(tpmCompositeHash((#select:=pm,pcrValue:=s`pcrs#)),n,
                           signed(private(key),clear)),
                  TPM_SUCCESS);

  %% Prove that skipping SENTER is detectable by showing the quote following
  %% a command differs when SENTER is excluded.
  %% proved - Tue Jun 12 15:34:24 CDT 2012
  no_senter: THEOREM
    FORALL (state:tpmAbsState,k:(tpmKey?),hv:HV,n:(tpmNonce?),pm:PCR_SELECTION,
ind:PCRINDEX) :
      runState(
         TPM_Init
         >> TPM_Startup(TPM_ST_CLEAR)
         >> CPU_senter
         >> CPU_sinit
         >> TPM_Extend(ind,hv)
         >> TPM_Quote(k,n,pm))
         (state)
      /=
      runState(
         TPM_Init
         >> TPM_Extend(ind,hv)
         >> TPM_Quote(k,n,pm))
         (state)

  TPM_PCR_Reset(pcrSelection:PCR_SELECTION):State =
    modifyOutput(
      (LAMBDA (s:tpmAbsState):executeCom(s,ABS_PCR_Reset(pcrSelection))),
      (LAMBDA (s:tpmAbsState):outputCom(s,ABS_PCR_Reset(pcrSelection))));

  pcr_reset_post : THEOREM
   FORALL(state:(afterStartup?),select:PCR_SELECTION) :
    LET (a,s) = runState(TPM_PCR_Reset(select))(state) IN
    IF null?(select)
    THEN a=OUT_Error(TPM_INVALID_PCR_INFO) AND s=state
    ELSE LET num=validatePCRVals(state,select) IN
```

```
        COND
          num=0 -> a=OUT_PCR_Reset(TPM_SUCCESS) AND
                    s=state WITH [`pcrs:=pcrsResetSelection(state`pcrs,select)],
          num=1 -> a=OUT_Error(TPM_NOTRESETABLE) AND s=state,
          num=2 -> a=OUT_Error(TPM_NOTLOCAL) AND s=state,
          ELSE -> a=OUT_Error(TPM_SUCCESS)
        ENDCOND
    ENDIF;

  pcr_reset_post2 : THEOREM
   FORALL(state:(afterStartup?),select:PCR_SELECTION) :
    LET (a,s) = runState(TPM_PCR_Reset(select))(state) IN
    not null?(select) =>
      not a=OUT_Error(TPM_SUCCESS)

END tpm
```

7.2 Data Theory

```
%% ----
%%
%% Data Theory
%%
%% Description: Abstract data definition
%%
%% Dependencies:
%%  pcr.pvs
%%  authdata.pvs
%%  keyData.pvs
%%
%% ----

data [DVAL,HVAL:TYPE+] : THEORY

  BEGIN

  IMPORTING pcr[HVAL];
  IMPORTING authdata;
  IMPORTING keydata;
  IMPORTING types;


  %% Data items that the TPM is aware of
  tpmData : DATATYPE
  BEGIN
    %% Stub for migrateScheme so can take digest - createMigBlob    (4.4)
    tpmMigScheme(migScheme:migrateScheme) : tpmMigScheme?
```

```
  %% EK type- indicates what type of information the EK's dealing with    (4.11)
  tpmEKBlobActivate(sessK:(tpmSessKey?),idDigest:(tpmDigest?),
      pcrInfo:(tpmPCRInfoShort?)) : tpmEKBlobActivate?

  %% Abitrarily long digest of arbitrary TPM data    (5.4)
  tpmDigest(digest:list[tpmData],crs:CRYPTOSTATUS) : tpmDigest?
  %  tpmDigest is the list of things concatenated and hashed to create the
  %  digest value - #(d0++d1++...++dn).
  %  Note that this digest does not contain PCRs

  % TPM_COMPOSITE_HASH should be element of tpmDigest,
  % PCR_COMPOSITE : [#select:PCR_SELECTION,pcrValue:PCRVALUES#]
  tpmCompositeHash(dig:PCR_COMPOSITE) : tpmCompositeHash?

  %% Random value that provides protection from replay.    (5.5)
  tpmNonce(i:int) : tpmNonce?

  %% Authdata - don't know what it is yet    (5.6)
  tpmSecret(i:int) : tpmSecret?

  %% Provides proof that the associated public key has    (5.12)
  %% TPM Owner AuthData to be a migration key
  tpmMigKeyAuth(key:(tpmKey?),scheme:(tpmMigScheme?),digest:(tpmDigest?),
      crs:CRYPTOSTATUS) : tpmMigKeyAuth?

  %% PCR Info    (8.4)
  tpmPCRInfoLong(
      locAtCreation : LOCALITY %loc modifier when blob is created
      % Loc modifier req to reveal sealed data or use a key wrapped to PCRs
      % Value must not be zero (0)
      , locAtRelease  : LOCALITY
      % Selection of PCRs active when blob is created
      , creationPCRSelect : PCR_SELECTION
      % Selection of PCRs to which the key or data is bound
      , releasePCRSelect  : PCR_SELECTION
      % Composite digest value of the PCR values, when the blob is created
      , digAtCreation : (tpmCompositeHash?)
      % Digest of PCR indices and values to verify when revealing sealed data
      % or using a key that was wrapped to PCRs
      , digAtRelease  : (tpmCompositeHash?)
      ) : tpmPCRInfoLong?

tpmPCRInfoShort(
      pcrSelect : PCR_SELECTION
      % Selection of PCRs that specifies the digestAtRelease
      , locAtRelease : LOCALITY_SELECTION
      % Locality modifier required to release information
      % must not be zero (0)
      , digAtRelease : (tpmCompositeHash?)
```

```
    % Digest of PCR indices and PCR values to verify when revealing auth data
    ) : tpmPCRInfoShort?


%% Stored Data - necessary to ensure the enforcement of security
%% properties (9.2) Used by seal and unseal commands to identify
%% pcr index and values that must be present to properly unseal
%% data.
tpmStoredData(sealInfo:(tpmPCRInfoLong?),encData:(tpmSealedData?),
    crs:CRYPTOSTATUS) : tpmStoredData?
    %% encData is the piece encrypted by the crs.

%% Sealed Data - contains confidential info related to sealed data    (9.3)
tpmSealedData(authData:(tpmSecret?),tpmProof:(tpmSecret?),
    storedDigest:(tpmDigest?), data:tpmData,crs:CRYPTOSTATUS) : tpmSealedData?

%% Session keys are simply symetric keys    (9.4)
tpmSessKey(skey:KVAL,crs:CRYPTOSTATUS) : tpmSessKey?

%% Bound data    (9.5)
tpmBoundData(payloadData:tpmData,crs:CRYPTOSTATUS) : tpmBoundData?

%% Assymetric keys used by the TPM - wrapped, signed, encrypted    (10.3)
tpmKey(key:KVAL,keyUsage:KEY_USAGE,keyFlags:KEY_FLAGS,
    authDataUsage:AUTH_DATA_USAGE,algoParms:KEY_PARMS,
    PCRInfo:(tpmPCRInfoLong?),wrappingKey:KVAL,encDat:(tpmStoreAsymkey?),
    crs:CRYPTOSTATUS) : tpmKey?
%  tpmKey subsumes all asymmetric keys used by the TPM
%  encrypted key: tpmKey(k,...,encrypted(public(j),clear))
%  clear key encrypted with public j
%  signed/certified key: tpmKey(k,...,signed(private(j),clear))
%  clear key signed by private j

%% Pub Key    (10.5)
tpmPubkey(pubKey:KVAL):tpmPubkey?

%% Store Asym Key [private key]    (10.6)
tpmStoreAsymkey(usageAuth:(tpmSecret?),migrationAuth:(tpmSecret?),
    pubDataDigest:(tpmDigest?),privKey:KVAL,crs:CRYPTOSTATUS) : tpmStoreAsymkey?

%% Migrate Asym Key    (10.8)
tpmMigrateAsymkey(usageAuth:(tpmSecret?),pubDataDigest:(tpmDigest?),
    partPrivKey:KVAL) : tpmMigrateAsymkey?

%% Quote including a PCR digest and nonce.    (11.3)
tpmQuote(digest:(tpmCompositeHash?),externalData:(tpmNonce?),
    crs:CRYPTOSTATUS) : tpmQuote?
    %  PCRs are not current TPM data, so the digest is over PCR values.

%% Provides wrapper to each type of structure that will be in use when    (12.1)
```

```
    %% EK is in use
    tpmEKBlob(blob:tpmData,crs:CRYPTOSTATUS) : tpmEKBlob?
    %% blob must be tpmEKBlobAuth or tpmEKBlobActivate

    %% Certification request sent to Privacy CA    (12.5)
    tpmIdContents(digest:(tpmDigest?),aik:(tpmKey?),crs:CRYPTOSTATUS) : tpmIdContents?
    %  digest should contain CA public key, name, and AIK

    %% Contains symmetric key to encrypt the identity credential    (12.8)
    tpmAsymCAContents(sessK:(tpmSessKey?),idDigest:(tpmDigest?),
        crs:CRYPTOSTATUS) : tpmAsymCAContents?

  END tpmData;

  activateIdentityBlob?(blob:tpmData) : bool =
  CASES blob OF
    tpmEKBlob(b,c) : TRUE,
    tpmAsymCAContents(k,d,c) : TRUE
    ELSE FALSE
  ENDCASES

  tpmNonceZero : (tpmNonce?)
  tpmSealedNull : (tpmSealedData?)
  pcrInfoLongDefault : (tpmPCRInfoLong?)
  pcrInfoNull : (tpmPCRInfoLong?)
  storeAsymkeyDefault : (tpmStoreAsymkey?)
  tempAuthData : (tpmSecret?)

END data
```

7.3 PCR Theory

```
%% ----
%%
%% PCR Theory
%%
%% Description: Basic model of a TPM register file
%%
%% Dependencies:
%%  None
%%
%% ----

pcr [HV: TYPE+] : THEORY
  BEGIN

  %% Locality type - check with spec    (8.6)
```

```
  LOCALITY : TYPE = n:nat | n<=4;
  LOCALITY_SELECTION : TYPE = set[LOCALITY];

  %% PCR array index type - check with spec
  PCRINDEX : TYPE = n:nat | n <=23;

  %% Should be bit map that indicates if a PCR is active or not.    (8.1)
  PCR_SELECTION : TYPE = list[PCRINDEX];

  PCR : datatype
  begin
    reset : reset?
    resetOne : resetOne?
    extend(pcr:PCR,hash:HV) : extend?
  end PCR;

  PCRVALUES : TYPE = ARRAY[PCRINDEX->PCR];

  %% (8.2)
  PCR_COMPOSITE : TYPE = [#
   % The indication of which PCR values are active
     select : PCR_SELECTION
% Array of PCRVALUE structures.
% The values come in the order specified by the select parameter
%  and are concatenated into a single blob.
, pcrValue : PCRVALUES
   #];

  %% Information associated with PCR     (8.8)
  PCR_ATTRIBUTE : TYPE = [# pcrExtendLocal : LOCALITY_SELECTION
                                           , pcrResetLocal : LOCALITY_SELECTION
                                           , pcrReset : bool #];

  %% PCR flag array
  PCR_ATTRIBUTES : TYPE = ARRAY[PCRINDEX->PCR_ATTRIBUTE];

  allLocs : LOCALITY_SELECTION = add(0,add(1,add(2,add(3,add(4,emptyset)))));

  % (8.9)
  pcrDebug : PCR_ATTRIBUTES
  pcrInit : PCR_ATTRIBUTES

  % Pcr attributes are available on a per PCR basis. Differs by platform configuration
  allResetAccess : PCR_ATTRIBUTES =
    (LAMBDA (i:PCRINDEX) :
      (#
         pcrReset := true
       , pcrResetLocal := allLocs
       , pcrExtendLocal := allLocs
      #))
```

```
%% Some common PCR sets

%% Power on - unknown state
pcrsPower : PCRVALUES;

%% Following startup clear.  Nonresettable PCRs to 0 and resettable PCRs to -1
pcrsReset(flags:PCR_ATTRIBUTES) : PCRVALUES =
  (LAMBDA (i:PCRINDEX) :
    IF pcrReset(flags(i)) THEN resetOne ELSE reset ENDIF);

%% Following SENTER.  Nonresettable PCRs to same and resettable PCRs to 0
pcrsSenter(curr:PCRVALUES,flags:PCR_ATTRIBUTES) : PCRVALUES =
  (LAMBDA (i:PCRINDEX) :
    IF pcrReset(flags(i)) THEN reset ELSE curr(i) ENDIF);

pcrsResetSelection(curr:PCRVALUES,select:PCR_SELECTION) : PCRVALUES =
  (LAMBDA (i:PCRINDEX) :
    IF member(i,select) THEN reset ELSE curr(i) ENDIF);

pcrsResetSelectCorrect: THEOREM
 FORALL (pcrs:PCRVALUES,pm:PCR_SELECTION) :
   pcrsResetSelection(pcrs,pm)=
      (LAMBDA (i:PCRINDEX) : IF member(i,pm) THEN reset ELSE pcrs(i) ENDIF)


%% Some common PCR flag settings

%% PCR extension operator
pcrsExtend(pcrs:PCRVALUES,i:PCRINDEX,hv:HV) : PCRVALUES =
   (pcrs WITH [(i) := extend(pcrs(i),hv)])

getPCRs(pcrs:PCRVALUES,pcrMask:PCR_SELECTION) : list[PCR] =
   map(pcrs,pcrMask);

getPCRsCorrectness: THEOREM
(FORALL (pm:PCR_SELECTION, i:PCRINDEX, pcrs:PCRVALUES) :
   member(i,pm)
   =>
   member(pcrs(i),getPCRs(pcrs,pm)));

%% Theorems

%% Extension is antisymmetric
antisym : THEOREM FORALL (h1,h2:HV, p:PCR) :
  h1/=h2 => extend(p,h1) /= extend(p,h2)

END pcr
```

7.4 Key Theory

```
%% ----
%%
%% Key Theory
%%
%% Description: Basic model of keys, encryption, decryption, and signing
%%
%% Dependencies:
%%  data.pvs
%%  pcr.pvs
%%  keyData.pvs
%%
%% ----
key [DVAL,HVAL:TYPE+] : THEORY

  BEGIN

  IMPORTING data[DVAL,HVAL];

  %% Reserved Key Handles - Should be moved to tpm.pvs
  TPM_KH_SRK : (tpmKey?) = tpmKey(2,storage,keyFlagsF,
  always,keyParmsDef,
  pcrInfoLongDefault,1,
  storeAsymkeyDefault,clear)
  TPM_KH_OWNER : (tpmKey?)
  TPM_KH_REVOKE : (tpmKey?)
  TPM_KH_TRANSPORT : (tpmKey?)
  TPM_KH_OPERATOR : (tpmKey?)
  TPM_KH_ADMIN : (tpmKey?)
  %% Not sure if EK is an indentity key or not
  TPM_KH_EK : (tpmKey?) = tpmKey(1,identity,keyFlagsF,
 always,keyParmsDef,
 pcrInfoLongDefault,1,
 storeAsymkeyDefault,clear)


  %% Any tpmKey? with key value 0 is invalid.
  invalidKey?(k:(tpmKey?)):bool = key(k)=0;

  %% Define private and public key accessors for tpmKey?. The public
  %% key is the key value while the private key is -key.
  private(k:(tpmKey?)):KVAL = inverse(key(k))
  public(k:(tpmKey?)):KVAL = key(k)

  privateKey(k:(tpmKey?)):(tpmKey?) = tpmKey(inverse(key(k)),
     keyUsage(k),
     keyFlags(k),
```

```
   authDataUsage(k),
   algoParms(k),
   PCRInfo(k),
   wrappingKey(k),
   encDat(k),
   crs(k))
publicKey(k:(tpmKey?)):(tpmKey?) = k

%% Some theorems about key inverse, private and public.

%% The inverse key of the inverse key is the original key
double_inverse: THEOREM FORALL (k:KVAL) : inverse(inverse(k))=k;

%% the inverse of a private key is the public key for the same
%% asymmetric key.
inverse_private: THEOREM FORALL (k:(tpmKey?)) :
  inverse(private(k))=public(k);

%% visa versa
inverse_public: THEOREM FORALL (k:(tpmKey?)) :
  inverse(public(k))=private(k);

%% If inverse keys are equal, key values are equal
equal_inverse: THEOREM FORALL (k0,k1:KVAL) :
   inverse(k0) = inverse(k1) <=> k0=k1;

%% Basic crypto functions operating on KVAL xStatus functions
%% operate on status indicators while x functions operate on
%% tpmData.  Users need only call encrypt, decrypt, sign, checkSig
%% and friends.

% Data Encryption

encryptStatus(k:KVAL,c:CRYPTOSTATUS) : (encrypted?) =
  encrypted(k,c);

encrypt(k:KVAL,d:tpmData) : tpmData =
  CASES d OF
    tpmDigest(digest0,crs0) :
      tpmDigest(digest0,encryptStatus(k,crs0)),
    tpmMigKeyAuth(k0,s0,d0,crs0) :
      tpmMigKeyAuth(k0,s0,d0,encryptStatus(k,crs0)),
    tpmStoredData(s0,e0,crs0) : tpmStoredData(s0,e0,encryptStatus(k,crs0)),
    tpmSealedData(a0,p0,s0,d0,crs0) :
     tpmSealedData(a0,p0,s0,d0,encryptStatus(k,crs0)),
    tpmSessKey(k0,crs0) : tpmSessKey(k0,encryptStatus(k,crs0)),
    tpmBoundData(p0,crs0) : tpmBoundData(p0,encryptStatus(k,crs0)),
    tpmKey(k0,u0,f0,a0,ap0,p0,wk0,e0,crs0) :
      tpmKey(k0,u0,f0,a0,ap0,p0,wk0,e0,encryptStatus(k,crs0)),
    tpmStoreAsymkey(u0,m0,p0,k0,crs0) :
```

```
       tpmStoreAsymkey(u0,m0,p0,k0,encryptStatus(k,crs0)),
      tpmQuote(n0,p0,crs0) : tpmQuote(n0,p0,encryptStatus(k,crs0)),
      tpmEKBlob(b0,crs0) : tpmEKBlob(b0,encryptStatus(k,crs0)),
      tpmIdContents(digest0,aik0,crs0) :
        tpmIdContents(digest0,aik0,encryptStatus(k,crs0)),
      tpmAsymCAContents(s0,i0,crs0) :
       tpmAsymCAContents(s0,i0,encryptStatus(k,crs0))
      ELSE d
    ENDCASES;

  noCrypto?(d:tpmData) : bool =
    CASES d OF
      tpmMigScheme(m0) : TRUE,
      tpmEKBlobActivate(k0,i0,p0) : TRUE,
      tpmNonce(i0) : TRUE,
      tpmSecret(i0) : TRUE,
      tpmPCRInfoLong(lc0,lr0,sc0,sr0,dc0,dr0) : TRUE,
      tpmPubkey(k0) : TRUE,
      tpmMigrateAsymkey(u0,d0,p0) : TRUE
      ELSE FALSE
    ENDCASES;

  % Data decryption
  decryptStatus(k:KVAL,c:(encrypted?)) : CRYPTOSTATUS =
    IF k = inverse(key(c)) THEN crstat(c) ELSE c ENDIF;

  decrypt(k:KVAL,d:tpmData) : tpmData =
    CASES d OF
      tpmDigest(digest0,crs0) :
        IF encrypted?(crs0)
        THEN tpmDigest(digest0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmMigKeyAuth(k0,s0,d0,crs0) :
        IF encrypted?(crs0)
        THEN tpmMigKeyAuth(k0,s0,d0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmStoredData(s0,e0,crs0) :
        IF encrypted?(crs0)
        THEN tpmStoredData(s0,e0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmSealedData(a0,p0,s0,d0,crs0) :
        IF encrypted?(crs0)
        THEN tpmSealedData(a0,p0,s0,d0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmSessKey(k0,crs0) :
       IF encrypted?(crs0)
```

```
        THEN tpmSessKey(k0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmBoundData(p0,crs0) :
       IF encrypted?(crs0)
       THEN tpmBoundData(p0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmKey(k0,u0,f0,a0,ap0,p0,wk0,e0,crs0) :
       IF encrypted?(crs0)
       THEN tpmKey(k0,u0,f0,a0,ap0,p0,wk0,e0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmStoreAsymkey(u0,m0,p0,k0,crs0) :
        IF encrypted?(crs0)
        THEN tpmStoreAsymkey(u0,m0,p0,k0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmQuote(n0,p0,crs0) :
        IF encrypted?(crs0)
        THEN tpmQuote(n0,p0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmEKBlob(b0,crs0) :
        IF encrypted?(crs0)
THEN tpmEKBlob(b0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmIdContents(digest0,aik0,crs0) :
        IF encrypted?(crs0)
        THEN tpmIdContents(digest0,aik0,decryptStatus(k,crs0))
ELSE d
ENDIF,
      tpmAsymCAContents(k0,d0,crs0) :
        IF encrypted?(crs0)
THEN tpmAsymCAContents(k0,d0,decryptStatus(k,crs0))
ELSE d
ENDIF
      ELSE d
    ENDCASES;

  % Lemmas on decryption and encryption

  decrypt_encrypt : THEOREM FORALL (k:KVAL,c:CRYPTOSTATUS) :
    decryptStatus(inverse(k),encryptStatus(k,c)) = c;

  decrypt_equal_keys: THEOREM FORALL (k0,k1:KVAL,b:CRYPTOSTATUS) :
    k0=k1 IMPLIES decryptStatus(inverse(k1),encryptStatus(k0,b)) = b;

  no_decrypt_unequal_keys: THEOREM FORALL (k0,k1:KVAL,b:CRYPTOSTATUS) :
```

```
  k0/=k1 IMPLIES decryptStatus(inverse(k1),encryptStatus(k0,b)) = encryptStatus(k0,b)

% Data signing

signStatus(k:KVAL,c:CRYPTOSTATUS) : (signed?) =
  signed(k,c);

sign(k:KVAL,d:tpmData) : tpmData =
  CASES d OF
    tpmDigest(digest0,crs0) :
      tpmDigest(digest0,signStatus(k,crs0)),
    tpmMigKeyAuth(k0,s0,d0,crs0) :
      tpmMigKeyAuth(k0,s0,d0,signStatus(k,crs0)),
    tpmStoredData(s0,e0,crs0) :
     tpmStoredData(s0,e0,signStatus(k,crs0)),
    tpmSealedData(a0,p0,s0,d0,crs0) :
     tpmSealedData(a0,p0,s0,d0,signStatus(k,crs0)),
    tpmSessKey(k0,crs0) :
      tpmSessKey(k0,signStatus(k,crs0)),
    tpmBoundData(p0,crs0) :
      tpmBoundData(p0,signStatus(k,crs0)),
    tpmKey(k0,u0,f0,a0,ap0,p0,wk0,e0,crs0) :
      tpmKey(k0,u0,f0,a0,ap0,p0,wk0,e0,signStatus(k,crs0)),
    tpmStoreAsymkey(u0,m0,p0,k0,crs0) :
      tpmStoreAsymkey(u0,m0,p0,k0,signStatus(k,crs0)),
    tpmQuote(n0,p0,crs0) : tpmQuote(n0,p0,signStatus(k,crs0)),
    tpmEKBlob(b0,crs0) : tpmEKBlob(b0,signStatus(k,crs0)),
    tpmIdContents(digest0,aik0,crs0) :
      tpmIdContents(digest0,aik0,signStatus(k,crs0)),
    tpmAsymCAContents(k0,d0,crs0) :
      tpmAsymCAContents(k0,d0,signStatus(k,crs0))
    ELSE d
  ENDCASES;

% Data signature checking

checkSigStatus(k:KVAL,c:CRYPTOSTATUS) : bool =
  CASES c OF
    signed(kq,cq) : k = kq
    ELSE FALSE
  ENDCASES;

checkSig(k:KVAL,d:tpmData) : bool =
  CASES d OF
    tpmMigScheme(m0) : FALSE,
    tpmEKBlobActivate(k0,i0,p0) : FALSE,
    tpmCompositeHash(d0) : FALSE,
    tpmNonce(i0) : FALSE,
    tpmSecret(i0) : FALSE,
    tpmPCRInfoLong(lc0,lr0,sc0,sr0,dc0,dr0) : FALSE,
```

```
      tpmPCRInfoShort(ps0,l0,d0) : FALSE,
      tpmPubkey(k0) : FALSE,
      tpmMigrateAsymkey(u0,d0,p0) : FALSE
      ELSE
        IF signed?(crs(d))
       THEN checkSigStatus(k,crs(d))
ELSE FALSE
ENDIF
   ENDCASES;

  % Lemmas on data signing and signature checking

  check_sign : THEOREM FORALL (k:KVAL,c:CRYPTOSTATUS) :
    checkSigStatus(k,signStatus(k,c));

  %%%% Key sets, installation, and use

  %% A keyset is a set of key values
  KEYSET : TYPE = set[KVAL];

  %% Is a key installed or the SRK?
  installedOrSRK?(k:(tpmKey?),ks:KEYSET):bool =
    member(key(k),ks) OR key(k)=key(TPM_KH_SRK);


  %% There is no magic for SRK - use the TPM_KH_SRK handle if the wrapping
  %% key is the SRK.
  addKey(k:(tpmKey?),ks:KEYSET):KEYSET = add(key(k),ks);
    % IF (wrappingKey(k)=TPM_KH OR member(wrappingKey(k),ks)) AND d=PCRInfo(k)
    % THEN add(key(k),ks)
    % ELSE ks
    % ENDIF;

  loadKey(k:(tpmKey?),wk:(tpmKey?),ks:KEYSET,d:PCRVALUES):KEYSET =  %d:list[PCR]
    IF
%       getPCRs(d,releasePCRSelect(PCRInfo(k)))=digAtRelease(PCRInfo(k)) AND
%       getPCRs(d,creationPCRSelect(k))=digAtCreation(PCRInfo(k)) AND
      installedOrSRK?(wk,ks) %todo: should this be not?
    THEN addKey(k,ks)
    ELSE ks
    ENDIF

  load_key :THEOREM FORALL(k,w:(tpmKey?),ks:KEYSET,d:PCRVALUES) :
    loadKey(k,w,ks,d)=
    IF member(key(w),ks)
    THEN add(key(k),ks)
    ELSIF key(w)=key(TPM_KH_SRK)
    THEN add(key(k),ks)
    ELSE ks
    ENDIF
```

```
  %% Remove a key - this is not currently
  revokeKey(k:(tpmKey?),ks:KEYSET):KEYSET = remove(key(k),ks);

  child_if_parent: THEOREM FORALL (k,wk:(tpmKey?),srk:KVAL,ks:KEYSET,d:PCRVALUES) : %d:list[PCR]
    wrappingKey(k)=key(wk) AND storage?(keyUsage(wk))
    % AND d=PCRInfo(k)
    AND (member(wrappingKey(k),ks) OR wrappingKey(k)=key(TPM_KH_SRK))
       => member(key(k),loadKey(k,wk,ks,d))


  no_child_if_no_parent: THEOREM FORALL (k:(tpmKey?),ks:KEYSET) :
    NOT(member(key(k),revokeKey(k,ks)));

  %% Crypto functions using TPM keys.  Public keys from TPM keys are
  %% public knowledge and require nothing for use - encrypt and
  %% checkSig do not require a key set.  Private keys must be
  %% installed in the key set ks passed to each function - decrypt and
  %% sign require a key set.

  encryptWrapped(k:(tpmKey?),d:tpmData):tpmData = encrypt(key(k),d)

  decryptWrapped(k:(tpmKey?),d:tpmData,ks:KEYSET):tpmData =
    IF installedOrSRK?(k,ks) THEN decrypt(key(k),d) ELSE d ENDIF

  signWrapped(k:(tpmKey?),d:tpmData,ks:KEYSET):tpmData =
    IF installedOrSRK?(k,ks) THEN sign(key(k),d) ELSE d ENDIF

  checkSigWrapped(k:(tpmKey?),d:tpmData):bool = checkSig(key(k),d)

END key
```

7.5 Key Data Theory

```
%% ----
%%
%% keyData Theory
%%
%% Description: Detailed data of keys
%%
%% Dependencies:
%%  None
%%
%% ----
```

```
keydata : THEORY

  BEGIN

  KEYDATA : TYPE;

  %% Keys are integers.
  KVAL : TYPE = integer;

  %% The inverse of a key its negation
  inverse(k:KVAL):KVAL = -k;

  %% The bad key value is 0
  badkey:KVAL = 0;

  goodkey?(key:KVAL) : bool =
    key/=badkey

  %% Purpose for key

  %%    (5.8)
  KEY_USAGE : DATATYPE
  BEGIN
    signing : signing?
    storage : storage?
    identity : identity?
    authChange : authChange?
    bind : bind?
    legacy : legacy?
    migrate : migrate?
  END KEY_USAGE;

  %% Flags for defining key properties    (5.10)
  KEY_FLAGS : TYPE = [#redirection : bool
                                   , migratable : bool
                                   , isVolatile : bool
                                   , pcrIgnoredOnRead : bool
                                   , migrateAuthority : bool
                                   #];

  %% All key flags false
  keyFlagsF : KEY_FLAGS = (#redirection:= FALSE
                                         , migratable := FALSE
                                         , isVolatile := FALSE
                                         , pcrIgnoredOnRead := FALSE
                                         , migrateAuthority := FALSE #)

  ALGO_ID : DATATYPE
  BEGIN
    RSA : RSA?
```

```
    SHA : SHA?
    HMAC : HMAC?
    AESxxx : AES?
    MGF1 : MGF1?
  END ALGO_ID;

  KEY_PARMS : TYPE = [#algoId : ALGO_ID
%       , encScheme :
%       , sigScheme :
%       , parms :
              #];
  keyParmsDef : KEY_PARMS = (#algoId:=RSA#)

  %% Migragion scheme indicator    (4.10)
  migrateScheme : DATATYPE
  BEGIN
    migrate : migrate?
    rewrap : rewrap?
  END migrateScheme;

  %% Type indicating crypographic status - clear, encrypted, signed, wrapped,
  %% sealed.  Type is recursive allowing multiple operations on the same
  %% element.
  CRYPTOSTATUS : DATATYPE
  BEGIN
    clear : clear?
    encrypted(key:KVAL,crstat:CRYPTOSTATUS) : encrypted?
    signed(key:KVAL,crstat:CRYPTOSTATUS) : signed?
    sealed(key:KVAL,crstat:CRYPTOSTATUS) : sealed?
  END CRYPTOSTATUS;

END keydata
```

7.6 StateMonad Theory

```
%% ----
%%
%% StateMonad Theory
%%
%% Description: State monad model
%%
%% Dependencies:
%%  None
%%
%% ----

StateMonad[A,S:TYPE+] : THEORY
BEGIN
```

```
State : DATATYPE
BEGIN
  state(runState:[S->[A,S]]):state?
END State

%% return :: a -> M a -- Nothing special here
%%
%% x - output produced in the state
%% s - the state being associated with x
%%
%% 1. x is bound to a new output of type A
%% 2. state encapsulates a function of type S->[A,S]
%% 3. the value for A in runState is x, the input to return
%% 4. the value for S is held abstract until runState is applied
%% 5. whatever state is input to runState, the output will be the value
%% bound to x in return(x)
return(x:A):State = state(LAMBDA (s:S) : (x,s));

%% bind :: M a -> (a -> M b) -> M b -- For our purposes a=b making this
%% bind :: M a -> (a -> M a) -> M a
%%
%% m - State[A,S] - a runState function
%% f - [A->State[A,S]] - takes an A to a State[A,S]
%%
%% 1. LET cases the runState function to be extracted from m and calls it on
%% s0 resulting in a value of type [A,S].  s0 is the input state
%% 2. a is bound to the A element and s1 to the S element.  Thus, a is the
%% resulting output and s1 is the resulting state
%% 3. f transforms A into a State[A,S] containing a function of type S->[A,S]
%% 4. runState extracts the function and applies it to s1, the state resulting
%% from the invocation caused by the LET binding.
%%

>>= (m:State,f:[A->State]):State =
  state(LAMBDA(s0:S):
         LET (a,s1) = runState(m)(s0) IN
           runState(f(a))(s1));

>> (m:State,s:State):State =
  state(LAMBDA(s0:S):
         LET (a,s1) = runState(m)(s0) IN
           runState(s)(s1));

%% Monad laws -- might as well, we're in a prover

% Left identity -- bind(return(a),f) = f(a)

left_identity: LAW FORALL (a:A,f:[A->State]) : return(a) >>= f = f(a)
```

```
% Right identity -- bind(m,return) = m

right_identity: LAW FORALL (m:State) : m >>= return = m

% Associativity -- bind(bind(m,f),g) = bind(m,bind(lambda:x->f(x),g))

assocativity: LAW FORALL (m:State,f,g:[A->State]) :
  m >>= f >>= g = m >>= (lambda(x:A): f(x) >>= g)

%% Common Monadic functions

%% liftM lifts a state transformation into the state monad.  Given a function
%% from S to S, return a function from State to State.

liftM(f:[S->S]):[State->State] =
   (LAMBDA (m0:State) :
      state(LAMBDA (s0:S) :
              LET (a1,s1) = runState(m0)(s0) IN
                (a1,f(s1))));

%% ifM takes a predicate and a condition monad, evaluates the condition
%% monad and calls the predicate on the output.  Based on the predicate
%% value, one of two monads results.  ifM_ works the same, but ignores
%% the prior output.

%% Use with >>=
ifM(p:[A->bool],c,t,e:[A->State]): [A->State] =
   (LAMBDA (a:A) : c(a) >>= IF p(a) THEN t ELSE e ENDIF);

%% Use with >>
ifM_(p:[A->bool],c:State,t,e:[A->State]): State =
   c >>= (LAMBDA (a:A): IF p(a) THEN t(a) ELSE e(a) ENDIF);

%% Predefined, useful values for f.

% Replace state
put(a:A,s1:S) : State = state(LAMBDA(s0:S):(a,s1))

% Modify state using only the current state
modify(a:A,f:[S->S]) : State = state(LAMBDA(s0:S):(a,f(s0)))

% Generate output with no state modification
output(g:[S->A]) : State = state(LAMBDA(s0:S):(g(s0),s0))

% Modify state and generate output
modifyOutput(f:[S->S],g:[S->A]) : State =
    state(LAMBDA(s0:S):(g(s0),f(s0)))

% Modify state with previous output and current state.
useOutputState(f:[A->[S->S]]) : [A->State] =
```

```
    (LAMBDA (a:A):state(LAMBDA(s0:S):(a,f(a)(s0))))

 % Generate output with previous output and current state.
 useOutputOutput(f:[A->[S->A]]) : [A->State] =
    (LAMBDA (a:A):state(LAMBDA(s0:S):(f(a)(s0),s0)))

 % Generate an entire new state with previous output and current state.
 useOutputStateOutput(f:[A->[S->S]],g:[A->[S->A]]) : [A->State] =
    (LAMBDA (a:A):state(LAMBDA(s0:S):(g(a)(s0),f(a)(s0))))

END StateMonad
```

7.7 Memory Theory

```
%% ----
%%
%% Memory Theory
%%
%% Description: Basic model of memory
%%
%% Dependencies:
%%  None
%%
%% ----

memory [ C:type+,e:C ]
: THEORY

  BEGIN

  % ASSUMING
   % assuming declarations
  % ENDASSUMING

  % memory type
  mem : TYPE = [nat -> C];

  %% initial empty memory
  empty : mem = (LAMBDA (i:nat) : e);

  %% update a memory location
  updateLoc(m:mem,i:nat,v:C):mem =
    (LAMBDA (x:nat) : IF x=i THEN v ELSE m(x) ENDIF);

  %% empty contains nothing
  empty_empty: THEOREM FORALL (x:nat) : empty(x)=e;

  %% Updated memory returns the right value for the udpated location
```

```
  update1: THEOREM FORALL (m:mem,i:nat,c:C) :
    (updateLoc(m,i,c)(i)=c);

  %% Updated memoery returns the old value for everyting except the updated
  %% location
  update2: THEOREM FORALL (m:mem,i,j:nat,c:C) :
    i/=j => updateLoc(m,i,c)(j)=m(j);

  END memory
```

7.8 Startup Data Theory

```
%% ----
%%
%% startupData Theory
%%
%%
%% Description: Startup types, save, and restore
%%
%% Dependencies:
%%  PermanentData.pvs
%%  PermanentFlags.pvs
%%  StanyData.pvs
%%  StanyFlags.pvs
%%  StclearData.pvs
%%  StclearFlags.pvs
%%  key.pvs
%%  data.pvs
%%  keyData.pvs
%%  pcr.pvs
%%  authdata.pvs
%%
%% ----

startupData[B,K,HV : TYPE+] : THEORY

  BEGIN

  IMPORTING key[B,HV];
  IMPORTING PermanentFlags;
  IMPORTING PermanentData[B,K,HV];
  IMPORTING StanyFlags[HV];
  IMPORTING StanyData[B,HV];
  IMPORTING StclearFlags;
  IMPORTING StclearData[B,HV];


  %% Flags used by the TPM_Startup command    (4.5)
```

```
TPM_STARTUP_TYPE : DATATYPE
BEGIN
  TPM_ST_CLEAR : TPM_ST_CLEAR?
  TPM_ST_STATE : TPM_ST_STATE?
  TPM_ST_DEACTIVATED : TPM_ST_DEACTIVATED?
END TPM_STARTUP_TYPE

%% Record containing state data to be stored on TPM_SaveState invocation
%% and restored when using TPM_ST_STATE flag.  The valid? flag indicates
%% if whether the record is valid or not.  Note that EK and SRK should
%% technically be in the permData structure, but we like to keep those
%% values exposed at the TPM state level.  Most presentations of the TPM
%% do this and do not reference permData or permFlags.  Thus, we try
%% to be consistent here.
restoreStateData : TYPE = [#
                        valid? : bool,
                        srk : (tpmKey?),
                        ek : (tpmKey?),
                        keyGenCnt : K,
                        keys : KEYSET,
                        pcrs : PCRVALUES,
                        permFlags : PermFlags,
                        permData : PermData,
                        stanyFlags: StanyFlags,
                        stanyData: StanyData,
                        stclearFlags: StclearFlags,
                        stclearData: StclearData
                        #];

wellFormedRestore?(r:restoreStateData) : bool =
  valid?(r) =>
  FORALL (i:PCRINDEX) :
    pcrReset(pcrAttrib(permData(r))(i)) => pcrs(r)(i) = resetOne;

%% Create a save record with keys, pcrs, and pcr flags.  Throw out pcr
%% values if they are resettable
saveState(tpmKeys: KEYSET,
          tpmEK : (tpmKey?),
  tpmSRK : (tpmKey?),
          tpmKeyGenCnt : K,
  tpmPcrs : PCRVALUES,
  tpmPermFlags : PermFlags,
  tpmPermData : PermData,
  tpmStanyFlags: StanyFlags,
  tpmStanyData: StanyData,
  tpmStclearFlags: StclearFlags,
  tpmStclearData: StclearData ) : (wellFormedRestore?) =
  (# valid? := TRUE,
     keys := tpmKeys,
     ek := tpmEK,
```

```
      srk := tpmSRK,
      keyGenCnt := tpmKeyGenCnt,
      pcrs := (LAMBDA (i:PCRINDEX) :
                   IF pcrReset(pcrAttrib(tpmPermData)(i))
                   THEN resetOne
                   ELSE tpmPcrs(i)
                   ENDIF),
      permFlags := tpmPermFlags,
      permData := tpmPermData,
      stanyFlags := tpmStanyFlags,
      stanyData := tpmStanyData,
      stclearFlags := tpmStclearFlags,
      stclearData := tpmStclearData
      #);

  %% Initial value for saved data.  The only field that matters is the valid?
  %% field that must be false.

  ekInit : (tpmKey?)
  srkInit : (tpmKey?)
  keyGenCntInit : K

  initSaveData : (wellFormedRestore?) =
    (# valid? := FALSE,
       keys := emptyset,
       ek := ekInit,
       srk := srkInit,
       keyGenCnt := keyGenCntInit,
       pcrs := pcrsPower,
       permFlags := permFlagsDefault,
       permData := permDataInit,
       stanyFlags:= stanyFlagsInit,
       stanyData := stanyDataInit,
       stclearFlags:= stclearFlagsInit,
       stclearData := stclearDataInit
    #);

  END startupData
```

7.9 Permanent Data Theory

```
%% ----
%%
%% Permanent Data Theory
%%
%% Description:
%%  This structure contains the data fields that are permanently held in the TPM
%%   and not affected by TPM_Startup(any)
```

```
%%
%% Dependencies:
%%  data.pvs
%%  pcr.pvs
%%  keyData.pvs
%%
%% ----

PermanentData[DV,KV,HV:TYPE+] : THEORY
  BEGIN

  IMPORTING data[DV,HV];

  PermData : TYPE = [#
    %% This is a random number that each TPM maintains to validate blobs in the
    %%  SEAL and other processes. THe default value is manufacturer specific.
      tpmProof : (tpmSecret?)
    %%
    , ekReset : (tpmNonce?)
    %% This is the TPM-Owner's AuthData data. default is manufacturer-specific.
    , ownerAuth : (tpmSecret?)
    %% The value that allows the execution of the TPM_SetTempDeactivated command
    , operatorAuth : (tpmSecret?)
    %% PCR attributes set during manufacturing for specific register, the
    %%  attributes must match requirements of the TCG platform specific
    %%  specification that describes the platform.
    , pcrAttrib : PCR_ATTRIBUTES
  #];

  nonce:(tpmNonce?)
  INVALIDPROOF:nat
  INVALIDAUTH:nat
  permDataInit : PermData = (# tpmProof:=tpmSecret(INVALIDPROOF)
                             , ekReset:=nonce
                             , ownerAuth:=tpmSecret(INVALIDAUTH)
                             , operatorAuth:=tpmSecret(INVALIDAUTH)
                             , pcrAttrib := allResetAccess
                             #)

  END PermanentData
```

7.10 Permanent Flags Theory

```
%% ----
%%
%% Permanent Flags Theory
%%
%% Description:
```

```
%%  These flags maintain state information for the TPM. The values are not
%%   affected by any TPM_Startup command.
%%
%% Dependencies:
%%  none
%%
%% ----

PermanentFlags : THEORY
  BEGIN

  PermFlags : TYPE = [#
    % The state of the disable flag. The default state is TRUE
      disable : bool
    % The ability to install an owner. The default state is TRUE
    , ownership : bool
    % The state of the inactive flag. The default state is TRUE
    , deactivated : bool
    % The ability to read the PUBEK without owner AuthData. default TRUE.
    , readPubek : bool
    % Whether the owner authorized clear commands are active. default FALSE.
    , disableOwnerClear : bool
    % Whether the TPM Owner may create a maintenance archive. default TRUE if
    %  maintenance is implemented, vendor specific if maintenance is not
    %  implemented.
    , allowMaintenance : bool
    % This bit can only be set to TRUE; it cannot be set to FALSE except during
    %  the manufacturing process.
    % FALSE: The state of either physicalPresenceHWEnable or
    %  physicalPresenceCMDEnable MAY be changed. (DEFAULT)
    % TRUE: The state of either physicalPresenceHWEnable or
    %  physicalPresenceCMDEnable MUST NOT be changed for the life of the TPM.
    , physicalPresenceLifetimeLock : bool
    % FALSE: Disable the hardware signal indicating physical presence. (DEFAULT)
    % TRUE: Enables the hardware signal indicating physical presence.
    , physicalPresenceHWEnable : bool
    % FALSE: Disable the command indicating physical presence. (DEFAULT)
    % TRUE: Enables the command indicating physical presence.
    , physicalPresenceCMDEnable : bool
    % TRUE: The PRIVEK and PUBEK were created using TPM_CreateEndorsementKeyPair
    % FALSE: The PRIVEK and PUBEK were created using a manufacturer's process.
    % NOTE: This flag has no default value as the key pair MUST be created by
    %  one or the other mechanism.
    , CEKPUsed : bool
    % TRUE: This TPM operates in FIPS mode
    % FALSE: This TPM does NOT operate in FIPS mode
    , FIPS : bool
    % TRUE: The operator AuthData value is valid
    % FALSE: the operator AuthData value is not set (DEFAULT)
    , operator : bool
```

```
  % TRUE: The TPM_RevokeTrust command is active
  % FALSE: the TPM RevokeTrust command is disabled
  , enableRevokeEK : bool
  % TRUE: All NV area authorization checks are active
  % FALSE: No NV area checks are performed, except for maxNVWrites. (DEFAULT)
  , nvLocked : bool
  % TRUE: GetPubKey will return the SRK pub key
  % FALSE: GetPubKey will not return the SRK pub key Default SHOULD be FALSE.
  %       See the informative.
  , readSRKPub : bool
  % TRUE: TPM_HASH_START has been executed at some time
  % FALSE: TPM_HASH_START has not been executed at any time (DEFAULT)
  % Reset to FALSE using TSC_ResetEstablishmentBit
  , tpmEstablished : bool
  % TRUE: A maintenance archive has been created for the current SRK
  , maintenanceDone : bool
  % TRUE: The full dictionary attack TPM_GetCapability info is deactivated.
  %  The returned structure is TPM_DA_INFO_LIMITED.
  % FALSE: The full dictionary attack TPM_GetCapability info is activated.
  %  The returned structure is TPM_DA_INFO. (DEFAULT)
  , disableFullDALogicInfo : bool
#];

disableDef:bool = TRUE
ownershipDef:bool = TRUE
deactivatedDef:bool = TRUE
readPubekDef:bool = TRUE
disableOwnerClearDef:bool = FALSE
allowMaintenanceDef:bool  = TRUE
physPresLLDef:bool = FALSE
physPresHWEDef:bool = FALSE
physPresCMDEDef:bool = FALSE
CEKPUsedDef:bool
FIPSDef:bool
operatorDef:bool = FALSE
enableRevokeEKDef:bool
nvLockedDef:bool = FALSE
readSRKPubDef:bool = FALSE
tpmEstablishedDef:bool= FALSE
maintenanceDoneDef:bool
disableFullDALogicInfoDef:bool = FALSE

permFlagsDefault : PermFlags = (#
    disable:=disableDef
  , ownership:=ownershipDef
  , deactivated:=deactivatedDef
  , readPubek:=readPubekDef
  , disableOwnerClear:=disableOwnerClearDef
  , allowMaintenance:=allowMaintenanceDef
  , physicalPresenceLifetimeLock:=physPresLLDef
```

```
    , physicalPresenceHWEnable:=physPresHWEDef
    , physicalPresenceCMDEnable:=physPresCMDEDef
    , CEKPUsed:=CEKPUsedDef
    , FIPS:=FIPSDef
    , operator:=operatorDef
    , enableRevokeEK:=enableRevokeEKDef
    , nvLocked:=nvLockedDef
    , readSRKPub:=readSRKPubDef
    , tpmEstablished:=tpmEstablishedDef
    , maintenanceDone:=maintenanceDoneDef
    , disableFullDALogicInfo:=disableFullDALogicInfoDef
  #);

  END PermanentFlags
```

7.11 Stany Data Theory

```
%% ----
%%
%% STANY Data Theory
%%
%% Description:   Most of the data in this structure resets on
%%  TPM_Startup(ST_STATE). A TPM may implement rules that provide longer-term
%%  persistence for the data. The TPM reflects how it handles the data in
%%  various TPM_GetCapability fields including startup effects.
%%
%% Dependencies:
%%  data.pvs
%%  keydata.pvs
%%  authdata.pvs
%%  pcr.pvs
%%
%% ----
StanyData[DV,HV:TYPE+] : THEORY
  BEGIN

  IMPORTING data[DV,HV];

  StanyData : TYPE = [#
   % The nonce used to properly identify saved session context blobs.
   % This MUST be set to all zeros on each TPM_Startup (ST_Clear).
   % The nonce MAY be set to all zeros on TPM_Startup(any).
   % Flag Name : TPM_AD_CONTEXTNONCESESSION
   contextNonceSession : (tpmNonce?)
   % This is the counter to avoid session context blob replay attacks.
   % This MUST be set to 0 on each TPM_Startup (ST_Clear). The value MAY
   %  be set to 0 on TPM_Startup (any).
   % Flag Name : TPM_AD_CONTEXTCOUNT
```

```
   , contextCount : int
   % This is the list of outstanding session blobs.
   % All elements of this array MUST be set to 0 on each
   % TPM_Startup(ST_Clear). The values MAY be set to 0 on
   % TPM_Startup(any). TPM_MIN_SESSION_LIST MUST be 16 or greater.
   % Flag Name : TPM_AD_CONTEXTLIST
   , contextList : int
  #];

  stanyDataInit : StanyData =
    (# contextNonceSession:=tpmNonceZero
    , contextCount:=0
    , contextList:=0
  #);

END StanyData
```

7.12 Stany Flags Theory

```
%% ----
%%
%% STANY Flags Theory
%%
%% Description:
%%  These flags reset on any TPM_Startup command. postInitialize indicates only
%%   that TPM_Startup has run, not that it was successful
%%  TOSPresent indicates the presence of a Trusted Operating System (TOS) that
%%   was established using the TPM_HASH_START command in the TPM Interface.
%%
%% Dependencies:
%%  pcr.pvs
%%
%% ----

StanyFlags[HV:TYPE+] : THEORY
  BEGIN

  IMPORTING pcr[HV];

  StanyFlags : TYPE = [#
      % Prevents the operation of most capabilities. There is no default
   % state. Initialized by TPM_Init to TRUE.TPM_Startup sets it to FALSE
       postInitialize : bool
   % This SHALL indicate for each command the presence of a locality
   %  modifier for the command. It MUST be always ensured that the value
   %  during usage reflects the current active locality.
   , localityModifier : LOCALITY
```

```
   % Defaults to FALSE
   % TRUE when there is an exclusive transport session active. Execution
   %  of ANY command other than TPM_ExecuteTransport targeting the
   %  exclusive transport session MUST invalidate the exclusive
   %  transport session.
   , transportExclusive : bool
   % Defaults to FALSE
   % Set to TRUE on TPM_HASH_START set to FALSE using setCapability
   , TOSPresent : bool
  #];

  stanyFlagsInit : StanyFlags =
    (# postInitialize:=FALSE,
               localityModifier:=0, %TODO: NOT CORRECT!
    transportExclusive:=FALSE,
    TOSPresent:=FALSE
  #);

  END StanyFlags
```

7.13 Stclear Data Theory

```
%% ----
%%
%% STCLEAR Data Theory
%%
%% Description:
%%  Most of the data in this structure resets on TPM_Startup(ST_CLEAR). A TPM
%%   may implement rules that provide longer-term persistence for the data. The
%%   TPM reflects how it handles the data in various TPM_GetCapablity fields
%%   including startup effects.
%%
%% Dependencies:
%%  key.pvs
%%  data.pvs
%%  keyData.pvs
%%  authdata.pvs
%%  pcr.pvs
%%
%% ----

StclearData[DV,HV:TYPE+] : THEORY
  BEGIN

  IMPORTING key[DV,HV];

    StclearData : TYPE = [#
   % This is the nonce in use to properly identify saved key context blobs.
```

```
   % This SHALL be set to all zeros on each TPM_Startup (ST_Clear).
   contextNonceKey : (tpmNonce?)
   % Points to where to obtain the owner secret in OIAP and OSAP
   %  commands. This allows a TSS to manage 1.1 applications on a 1.2
   %  TPM where delegation is in operation.
   % Default value is TPM_KH_OWNER.
   , ownerReference : int
   % Disables TPM_ResetLockValue upon authorization failure. The value
   %  remains TRUE for the timeout period. Default  FALSE.
   % The value is in the STCLEAR_DATA structure as the implementation of
   %  this flag is TPM vendor specific.
   , disableResetLock : bool
   % Platform configuration registers
   , PCR : PCRVALUES
   % The value can save the assertion of physicalPresence. Individual
   %  bits indicate to its ordinal that physicalPresence was previously
   %  asserted when the software state is such that it can no longer be
   %  asserted.
   % Set to zero on each TPM_Startup(ST_Clear).
   , deferredPhysicalPresence : int
  #];

  stclearDataInit : StclearData =
   (# contextNonceKey:=tpmNonceZero
          , ownerReference:=key(TPM_KH_OWNER)
          , disableResetLock:=FALSE
          , PCR:= pcrsPower
          , deferredPhysicalPresence:=0
 #);

END StclearData
```

7.14 Stclear Flags Theory

```
%% ----
%%
%% STCLEAR Flags Theory
%%
%% Description:
%%  These flags maintain state that is reset on teach TPM_Startup(ST_CLEAR)
%%  command. The values are not affected by TPM_Startup(ST_STATE) commands.
%%
%% Dependencies:
%%  none
%%
%% ----

StclearFlags : THEORY
```

```
  BEGIN

  StclearFlags : TYPE = [#
    % Prevents the operation of most capabilities. There is no default
    %  state. It is initialized by TPM_Startup to the same value as
    %  TPM_PERMANENT_FLAGS->deactivated or a set value depending on the
    %  type of TPM_Startup. TPM_SetTempDeactivated sets it to TRUE.
    deactivated : bool
    % Prevents the operation of TPM_ForceClear when TRUE. default FALSE.
    % TPM_DisableForceClear sets it to TRUE.
    , disableForceClear : bool
    % Command assertion of physical presence. default FALSE.
    %  This flag is affected by the TSC_PhysicalPresence command but not
    %  by the hardware signal.
    , physicalPresence : bool
    % Indicates whether changes to TPM_STCLEAR_FLAGS->physicalPresence
    %  flag are permitted.
    % TPM_Startup(ST_CLEAR) sets PhysicalPresenceLock to default (FALSE)
    %  (allow changes to the physicalPresence flag).
    %  When TRUE, the physicalPresence flag is FALSE.
    % TSC_PhysicalPresence can change the state of physicalPresenceLock.
    , physicalPresenceLock : bool
    % Set to FALSE on each TPM_Startup(ST_CLEAR). Set to TRUE when a
    %  write to NV_Index=0 is successful.
    , bGlobablLock : bool
   #];

  stclearFlagsInit : StclearFlags =
    (# deactivated:=TRUE
    , disableForceClear:=FALSE
    , physicalPresence:=FALSE
    , physicalPresenceLock:=FALSE
    , bGlobablLock:=FALSE
    #);

END StclearFlags
```

7.15 Return Codes Theory

```
%% ----
%%
%% Return Codes Theory
%%
%% Description: Datatype for Return Codes of TPM
%%
%% Dependencies:
%%  none
```

```
%%
%% ----

ReturnCodes : THEORY
  BEGIN

  %% Basic ReturnCode datatype
  ReturnCode : DATATYPE
  BEGIN
    TPM_SUCCESS : TPM_SUCCESS?
    TPM_VENDOR_ERROR : TPM_VENDOR_ERROR?
    %% Fatal Error
    % Authentication failed
    TPM_AUTHFAIL : TPM_AUTHFAIL?
    % The index to a PCR, DIR or other register is incorrect
    TPM_BADINDEX : TPM_BADINDEX?
    % One or more parameter is bad
    TPM_BAD_PARAMETER : TPM_BAD_PARAMETER?
    % Operation completed successfully but the auditing of that operation failed
    TPM_AUDITFAILURE : TPM_AUDITFAILURE?
    % The clear disable flag is set and all clear operations now require
    %  physical access.
    TPM_CLEAR_DISABLED : TPM_CLEAR_DISABLED?
    % The TPM is deactivated
    TPM_DEACTIVATED : TPM_DEACTIVATED?
    % The TPM is disabled
    TPM_DISABLED : TPM_DISABLED?
    % The target command has been disabled
    TPM_DISABLED_CMD : TPM_DISABLED_CMD?
    % The operation failed
    TPM_FAIL : TPM_FAIL?
    % The ordinal was unknown or inconsistent
    TPM_BAD_ORDINAL : TPM_BAD_ORDINAL?
    % The ability to install an owner is disabled
    TPM_INSTALL_DISABLED : TPM_INSTALL_DISABLED?
    % The key handle presented was invalid
    TPM_INVALID_KEYHANDLE : TPM_INVALID_KEYHANDLE?
    % The target key was not found
    TPM_KEYNOTFOUND : TPM_KEYNOTFOUND?
    % Unacceptable encryption scheme
    TPM_INAPPROPRIATE_ENC : TPM_INAPPROPRIATE_ENC?
    % Migration authorization failed
    TPM_MIGRATEFAIL : TPM_MIGRATEFAIL?
    % PCR information could not be interpreted
    TPM_INVALID_PCR_INFO : TPM_INVALID_PCR_INFO?
    % No room to load key.
    TPM_NOSPACE : TPM_NOSPACE?
    % There is no SRK set
    TPM_NOSRK : TPM_NOSRK?
    % An encrypted blob is invalid or was not created by this TPM
```

```
TPM_NOTSEALED_BLOB : TPM_NOTSEALED_BLOB?
% There is already an Owner
TPM_OWNER_SET : TPM_OWNER_SET?
% TPM has insufficient internal resources to perform the requested action.
TPM_RESOURCES : TPM_RESOURCES?
% A random string was too short
TPM_SHORTRANDOM : TPM_SHORTRANDOM?
% The TPM does not have the space to perform the operation.
TPM_SIZE : TPM_SIZE?
% The named PCR value does not match the current PCR value.
TPM_WRONGPCRVAL : TPM_WRONGPCRVAL?
% The paramSize argument to the command has the incorrect value
TPM_BAD_PARAM_SIZE : TPM_BAD_PARAM_SIZE?
% There is no existing SHA-1 thread.
TPM_SHA_THREAD : TPM_SHA_THREAD?
% The calculation is unable to proceed because the existing SHA-1 thread has
%  already encountered an error.
TPM_SHA_ERROR : TPM_SHA_ERROR?
% Self-test has failed and the TPM has shutdown.
TPM_FAILEDSELFTEST : TPM_FAILEDSELFTEST?
% The authorization for the 2nd key in a 2 key function failed authorization
TPM_AUTH2FAIL : TPM_AUTH2FAIL?
% The tag value sent to for a command is invalid
TPM_BADTAG : TPM_BADTAG?
% An IO error occurred transmitting information to the TPM
TPM_IOERROR : TPM_IOERROR?
% The encryption process had a problem.
TPM_ENCRYPT_ERROR : TPM_ENCRYPT_ERROR?
% The decryption process did not complete.
TPM_DECRYPT_ERROR : TPM_DECRYPT_ERROR?
% An invalid handle was used.
TPM_INVALID_AUTHHANDLE : TPM_INVALID_AUTHHANDLE?
% The TPM does not a EK installed
TPM_NO_ENDORSEMENT : TPM_NO_ENDORSEMENT?
% The usage of a key is not allowed
TPM_INVALID_KEYUSAGE : TPM_INVALID_KEYUSAGE?
% The submitted entity type is not allowed
TPM_WRONG_ENTITYTYPE : TPM_WRONG_ENTITYTYPE?
% The command was received in the wrong sequence relative to TPM_Init and a
%  subsequent TPM_Startup
TPM_INVALID_POSTINIT : TPM_INVALID_POSTINIT?
% Signed data cannot include additional DER information
TPM_INAPPROPRIATE_SIG : TPM_INAPPROPRIATE_SIG?
% The key properties in TPM_KEY_PARMs are not supported by this TPM
TPM_BAD_KEY_PROPERTY : TPM_BAD_KEY_PROPERTY?
% The migration properties of this key are incorrect.
TPM_BAD_MIGRATION : TPM_BAD_MIGRATION?
% The signature or encryption scheme for this key is incorrect or not
%  permitted in this situation.
TPM_BAD_SCHEME : TPM_BAD_SCHEME?
```

```
% The size of the data (or blob) parameter is bad or inconsistent with the
%  referenced key
TPM_BAD_DATASIZE : TPM_BAD_DATASIZE?
% A mode parameter is bad, such as capArea or subCapArea for
%  TPM_GetCapability, physicalPresence parameter for TPM_PhysicalPresence,
%  or migrationType for TPM_CreateMigrationBlob.
TPM_BAD_MODE : TPM_BAD_MODE?
% Either physicalPresence or physicalPresenceLock bits have the wrong value
TPM_BAD_PRESENCE : TPM_BAD_PRESENCE?
% The TPM cannot perform this version of the capability
TPM_BAD_VERSION : TPM_BAD_VERSION?
% The TPM does not allow for wrapped transport sessions
TPM_NO_WRAP_TRANSPORT : TPM_NO_WRAP_TRANSPORT?
% TPM audit construction failed and the underlying command was returning a
%  failure code also
TPM_AUDITFAIL_UNSUCCESSFUL : TPM_AUDITFAIL_UNSUCCESSFUL?
% audit construction failed and the underlying command was returning success
TPM_AUDITFAIL_SUCCESSFUL : TPM_AUDITFAIL_SUCCESSFUL?
% Attempt to reset a PCR register that does not have resettable attribute
TPM_NOTRESETABLE : TPM_NOTRESETABLE?
% Attempt to reset a PCR register that requires locality and locality
%  modifier not part of command transport
TPM_NOTLOCAL : TPM_NOTLOCAL?
% Make identity blob not properly typed
TPM_BAD_TYPE : TPM_BAD_TYPE?
% When saving context identified resource type doesn't match actual resource
TPM_INVALID_RESOURCE : TPM_INVALID_RESOURCE?
% The TPM attempting to execute a command only available when in FIPS mode
TPM_NOTFIPS : TPM_NOTFIPS?
% The command is attempting to use an invalid family ID
TPM_INVALID_FAMILY : TPM_INVALID_FAMILY?
% The permission to manipulate the NV storage is not available
TPM_NO_NV_PERMISSION : TPM_NO_NV_PERMISSION?
% The operation requires a signed command
TPM_REQUIRES_SIGN : TPM_REQUIRES_SIGN?
% Wrong operation to load an NV key
TPM_KEY_NOTSUPPORTED : TPM_KEY_NOTSUPPORTED?
% NV_LoadKey blob requires both owner and blob authorization
TPM_AUTH_CONFLICT : TPM_AUTH_CONFLICT?
% The NV area is locked and not writable
TPM_AREA_LOCKED : TPM_AREA_LOCKED?
% The locality is incorrect for the attempted operation
TPM_BAD_LOCALITY : TPM_BAD_LOCALITY?
% The NV area is read only and can't be written to
TPM_READ_ONLY : TPM_READ_ONLY?
% There is no protection on the write to the NV area
TPM_PER_NOWRITE : TPM_PER_NOWRITE?
% The family count value does not match
TPM_FAMILYCOUNT : TPM_FAMILYCOUNT?
% The NV area has already been written to
```

```
TPM_WRITE_LOCKED : TPM_WRITE_LOCKED?
% The NV area attributes conflict
TPM_BAD_ATTRIBUTES : TPM_BAD_ATTRIBUTES?
% The structure tag and version are invalid or inconsistent
TPM_INVALID_STRUCTURE : TPM_INVALID_STRUCTURE?
% The key is under control of TPM Owner and can only be evicted by TPM Owner
TPM_KEY_OWNER_CONTROL : TPM_KEY_OWNER_CONTROL?
% The counter handle is incorrect
TPM_BAD_COUNTER : TPM_BAD_COUNTER?
% The write is not a complete write of the area
TPM_NOT_FULLWRITE : TPM_NOT_FULLWRITE?
% The gap between saved context counts is too large
TPM_CONTEXT_GAP : TPM_CONTEXT_GAP?
% The maximum number of NV writes without an owner has been exceeded
TPM_MAXNVWRITES : TPM_MAXNVWRITES?
% No operator authorization value is set
TPM_NOOPERATOR : TPM_NOOPERATOR?
% The resource pointed to by context is not loaded
TPM_RESOURCEMISSING : TPM_RESOURCEMISSING?
% The delegate administration is locked
TPM_DELEGATE_LOCK : TPM_DELEGATE_LOCK?
% Attempt to manage a family other then the delegated family
TPM_DELEGATE_FAMILY : TPM_DELEGATE_FAMILY?
% Delegation table management not enabled
TPM_DELEGATE_ADMIN : TPM_DELEGATE_ADMIN?
% There was a command executed outside of an exclusive transport session
TPM_TRANSPORT_NOTEXCLUSIVE : TPM_TRANSPORT_NOTEXCLUSIVE?
% Attempt to context save a owner evict controlled key
TPM_OWNER_CONTROL : TPM_OWNER_CONTROL?
% The DAA command has no resources available to execute the command
TPM_DAA_RESOURCES : TPM_DAA_RESOURCES?
% The consistency check on DAA parameter inputData0 has failed.
TPM_DAA_INPUT_DATA0 : TPM_DAA_INPUT_DATA0?
% The consistency check on DAA parameter inputData1 has failed.
TPM_DAA_INPUT_DATA1 : TPM_DAA_INPUT_DATA1?
% The consistency check on DAA_issuerSettings has failed.
TPM_DAA_ISSUER_SETTINGS : TPM_DAA_ISSUER_SETTINGS?
% The consistency check on DAA_tpmSpecific has failed.
TPM_DAA_TPM_SETTINGS : TPM_DAA_TPM_SETTINGS?
% The atomic process indicated by the submitted DAA command is not the
%  expected process.
TPM_DAA_STAGE : TPM_DAA_STAGE?
% The issuer's validity check has detected an inconsistency
TPM_DAA_ISSUER_VALIDITY : TPM_DAA_ISSUER_VALIDITY?
% The consistency check on w has failed.
TPM_DAA_WRONG_W : TPM_DAA_WRONG_W?
% The handle is incorrect
TPM_BAD_HANDLE : TPM_BAD_HANDLE?
% Delegation is not correct
TPM_BAD_DELEGATE : TPM_BAD_DELEGATE?
```

```
  % The context blob is invalid
  TPM_BADCONTEXT : TPM_BADCONTEXT?
  % Too many contexts held by the TPM
  TPM_TOOMANYCONTEXTS : TPM_TOOMANYCONTEXTS?
  % Migration authority signature validation failure
  TPM_MA_TICKET_SIGNATURE : TPM_MA_TICKET_SIGNATURE?
  % Migration destination not authenticated
  TPM_MA_DESTINATION : TPM_MA_DESTINATION?
  % Migration source incorrect
  TPM_MA_SOURCE : TPM_MA_SOURCE?
  % Incorrect migration authority
  TPM_MA_AUTHORITY : TPM_MA_AUTHORITY?
  % Attempt to revoke the EK and the EK is not revocable
  TPM_PERMANENTEK : TPM_PERMANENTEK?
  % Bad signature of CMK ticket
  TPM_BAD_SIGNATURE : TPM_BAD_SIGNATURE?
  % There is no room in the context list for additional contexts
  TPM_NOCONTEXTSPACE : TPM_NOCONTEXTSPACE?
  %% TPM_NON_FATAL
  % The TPM is too busy to respond to the command immediately, but the
  %  command could be submitted at a later time
  TPM_RETRY : TPM_RETRY?
  % TPM_ContinueSelfTest has has not been run
  TPM_NEEDS_SELFTEST : TPM_NEEDS_SELFTEST?
  % The TPM is currently executing the actions of TPM_ContinueSelfTest because
  %  the ordinal required resources that have not been tested.
  TPM_DOING_SELFTEST : TPM_DOING_SELFTEST?
  % TPM is defending against dictionary attacks & is in some time-out period
  TPM_DEFEND_LOCK_RUNNING : TPM_DEFEND_LOCK_RUNNING?
END ReturnCode;

cpuReturn : DATATYPE
BEGIN
  CPU_ERROR : CPU_ERROR?
  CPU_SUCCESS : CPU_SUCCESS?
  CPU_DECRYPT_ERROR : CPU_DECRYPT_ERROR?
  CPU_QUOTE_ERROR : CPU_QUOTE_ERROR?
END cpuReturn;

END ReturnCodes
```

7.16 CA Protocol Theory

```
 %% Verification of the quote generation process from init through
 %% generation of a appraiser's quote.
 %%
 %% Acronym map:
 %% MLE - Measured Launch Environment
```

```
   %% TSS - Trusted Software Stack
   %% CR - Certification Request
   %% CA - Certificate Authority used to sign AIKs
   %%
   %% Memory map:
   %% 0 -> identity
   %% 1 -> CA cert
   %% 2 -> TPM quote
   %%

caProtocol [ B:TYPE+ % BLOB
           , HV:TYPE+ % Hash value
           , hash:[B->HV] % Hash function
         ] : THEORY

BEGIN

  IMPORTING tpm[B,HV,hash]

  makeCertActivate(state:tpmAbsState,k:(tpmKey?),d:(tpmDigest?)) : tpmAbsState =
   LET key=tpmKey(keyGenCnt(state),keyUsage(k),keyFlags(k),authDataUsage(k),
              algoParms(k),tpmPCRInfoLong(makeIdentityLocality,
                                locAtRelease(PCRInfo(k)),
                                creationPCRSelect(PCRInfo(k)),
                                releasePCRSelect(PCRInfo(k)),
                                tpmCompositeHash((#select:=creationPCRSelect(PCRInfo(k)),
                                                  pcrValue := pcrs(state) #)),
                                digAtRelease(PCRInfo(k))),
              wrappingKey(state`srk),
              tpmStoreAsymkey(tpmSecret(1),state`permData`tpmProof,
                              pubDataDigest(encDat(k)),privKey(encDat(k)),clear),
              clear) IN
   state WITH[`keyGenCnt := 2+keyGenCnt(state)
             ,`memory := updateLoc(updateLoc(state`memory,0,
                                     OUT_MakeIdentity(key,tpmIdContents(d,key,
                                             signed(--keyGenCnt(state),clear)),
                                             TPM_SUCCESS)),
                             1,OUT_Certify(key,tpmAsymCAContents(
                                           tpmSessKey(1+keyGenCnt(state),clear),
                                       d,encrypted(1,clear)),CPU_SUCCESS))]

  cert_and_quote_with_prev_key: THEOREM
    FORALL (state:(afterStartup?),n:(tpmNonce?),pm:PCR_SELECTION
           ,idKey:(tpmKey?),caDig:(tpmDigest?)) :
    LET (a,s) = runState(
% At this piont, the TSS holds an activation record
% encrypted by an EK.  If it is encrypted with its TPM's EK
% it can decrypt the signed AIK and session key.  The
% ActivateIdentity command does this.  Note that that is all
% ActivateIdentity does.  Nothing is installed and the TPM
```

```
% state is not modified in any way.
%          >>= (LAMBDA (a:tpmAbsOutput) :
%       CASES a OF
% % outIdentActivation(actc,sk,actek,rc) : TPM_ActivateIdentity(wkey(actc),k)
%  OUT_Certify(j,d,m) : TPM_ActivateIdentity(j,d,i3,i4)
%  ELSE TPM_Noop(a)
%       ENDCASES)
%  >>
CPU_read(0)
% Only the correct TPM can decrypt the certificate
% and session key.  This is critical as it links the
% certificate and session key to the TPM.  Now the TPM
% generates a quote signed by the AIK using the Quote
% command.
>>= (LAMBDA (a:tpmAbsOutput) :
     CASES a OF
OUT_MakeIdentity(ik,b,m) : TPM_Quote(ik,n,pm)
ELSE TPM_Noop(a)
     ENDCASES)
>>= CPU_saveOutput(2) % Quote is saved in 2 Now the
% TSS is ready to build the appraiser's quote.  It
% has the generated TPM quote signed by the AIK
% that only the TPM can generate. It has the
% certificate from the CA that only the TPM could
% decrypt.  It now ships the certificate and the quote
% back to the appraiser
>> CPU_BuildQuoteFromMem(2,0) % Build the quote
)
% When the appraiser recieves the quote, it can check the
% AIK signature and the AIK cert to ensure that the TPM
% quote came from a legitimate TPM associated with the AIK.
% However, it cannot collude with other appraisers to
% learn more about the target as they all can use different
% AIK values.
    (makeCertActivate(state,idKey,caDig)) IN
    % The start state here is the resulting state from running
    % TPM_MakeIdentity(d,k) >>= CPU_saveOutput(x) >>= CA_certify(k,i)
    %   >>= CPU_saveOutput(y) >>= TPM_ActivateIdentity(j,d)
    % as seen in the make_cert_activate_identity theorem from tpm.pvs
    LET key=idKey(s`memory(0)) IN
    makeIdentity?(state,idKey) AND
    OUT_MakeIdentity?(s`memory(0)) AND
    certify?(idKey(s`memory(0)),idBinding(s`memory(0))) AND
    OUT_Certify?(s`memory(1)) AND
    wellFormedRestore?(s`restore) AND
    activateIdentity?(tpmRestore(s`restore),idKey(s`memory(0)),dat(s`memory(1))) AND
    quote?(key) AND
    OUT_Quote?(s`memory(2))
    =>
    % The protocol works if the object being output is what we believe
```

```
 % should be output.
 a = OUT_FullQuote(
 % TPM quote
          tpmQuote(tpmCompositeHash((#select:=pm,pcrValue:=s‘pcrs#)),n,
             signed(private(key),clear)),
          % TODO
          tpmIdContents(caDig,key,signed(private(key),clear)),
          CPU_SUCCESS) AND
 s = state WITH[‘keyGenCnt := 2+keyGenCnt(state)
                ,‘memory := s‘memory]

ca_protocol:THEOREM
FORALL (state:(afterStartup?),d:(tpmDigest?),k:(tpmKey?),n:(tpmNonce?)
       ,p:PCR_SELECTION,x,y,z:nat) :
 LET (a,s) = runState(
                 TPM_MakeIdentity(d,k)
                 >>= CPU_saveOutput(x)
                 >>= (LAMBDA (a:tpmAbsOutput) :
                       CASES a OF
                         OUT_MakeIdentity(k,i,m) : CA_certify(k,i)
                         ELSE TPM_Noop(a)
                       ENDCASES)
                 >>= CPU_saveOutput(y)
                 >>= (LAMBDA (a:tpmAbsOutput) :
                       CASES a OF
                         OUT_Certify(j,d,m) : TPM_ActivateIdentity(j,d)
                         ELSE TPM_Noop(a)
                       ENDCASES)
                 >> CPU_read(x)
                 >>= (LAMBDA (a:tpmAbsOutput) :
                       CASES a OF
                         OUT_MakeIdentity(k,b,m) : TPM_Quote(k,n,p)
                         ELSE TPM_Noop(a)
                       ENDCASES)
                 >>= CPU_saveOutput(z)
                 >> CPU_BuildQuoteFromMem(z,x))
                 (state) IN
 LET key=idKey(s‘memory(x)) IN
 makeIdentity?(state,k) AND
 OUT_MakeIdentity?(s‘memory(x)) AND
 certify?(key,idBinding(s‘memory(x))) AND
 OUT_Certify?(s‘memory(y)) AND
 wellFormedRestore?(s‘restore) AND
 activateIdentity?(tpmRestore(s‘restore),key,dat(s‘memory(y))) AND
 quote?(key) AND
 OUT_Quote?(s‘memory(z))
 =>
 a=OUT_FullQuote(tpmQuote(tpmCompositeHash((#select:=p,pcrValue:=s‘pcrs#)),
                          n,signed(private(key),clear)),
                 tpmIdContents(d,key,signed(private(key),clear)),
```

```
                    CPU_DECRYPT_ERROR) AND
    s=state WITH [`memory:=s`memory
                  ,`keyGenCnt:=state`keyGenCnt+2]

END caProtocol
```

7.17 Invariants Theory

```
%% ----
%%
%% Invariants Theory
%%
%% Description: Prove invariants of TPM
%%
%% Dependencies:
%%  tpm.pvs
%%  StateMonad.pvs
%%  ReturnCodes.pvs
%%  memory.pvs
%%  StclearFlags.pvs
%%  startupData.pvs
%%  PermanentData.pvs
%%  StanyData.pvs
%%  StanyFlags.pvs
%%  key.pvs
%%  data.pvs
%%  keyData.pvs
%%  pcr.pvs
%%  authdata.pvs
%%  PermanentFlags.pvs
%%
%% ----

invariants [ B:TYPE+ % BLOB
      , HV:TYPE+ % Hash value
      , hash:[B->HV] % Hash function
     ] : THEORY

BEGIN

  IMPORTING tpm[B,HV,hash]

  restore_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
         ABS_SaveState?(c)) =>
        restore(s) = restore(executeCom(s,c));
```

```
memory_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_save?(c)) =>
      memory(s) = memory(executeCom(s,c));

srk_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_TakeOwnership?(c) OR
       ABS_OwnerClear?(c) OR
       ABS_ForceClear?(c) OR
       ABS_RevokeTrust?(c)) =>
      srk(s) = srk(executeCom(s,c));

ek_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
        ABS_CreateEndorsementKeyPair?(c) OR
       ABS_CreateRevocableEK?(c) OR
       ABS_RevokeTrust?(c) ) =>
      ek(s) = ek(executeCom(s,c));

keyGenCnt_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) or ABS_Init?(c) OR
       ABS_CreateWrapKey?(c) OR ABS_certify?(c) OR
       ABS_MakeIdentity?(c)) =>
      keyGenCnt(s) = keyGenCnt(executeCom(s,c));

keys_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
     ABS_LoadKey2?(c) OR
       ABS_ActivateIdentity?(c) OR
       ABS_OwnerClear?(c) OR
       ABS_ForceClear?(c) OR
       ABS_RevokeTrust?(c)) =>
    keys(s) = keys(executeCom(s,c));

% 1. The TPM SHALL only allow the following commands to alter the value
% of TPM_STCLEAR_DATA -> PCR
% a. TPM_Extend
% b. TPM_SHA1CompleteExtend
% c. TPM_Startup
% d. TPM_PCR_Reset
pcrs_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_sinit?(c) OR
```

```
       ABS_senter?(c) OR
       ABS_Extend?(c) OR
       ABS_PCR_Reset?(c)) =>
    pcrs(s) = pcrs(executeCom(s,c));

%% Monotonicity of locality
%% Assuming that we're not resetting or powering on, locality goes down
%% or remains the same
%% proved - Fri Sep 21 15:07:30 CDT 2012
monotonic_locality: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c)) =>
      locality(s) >= locality(executeCom(s,c));

%%%%% PermFlags
permFlags_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_SetOwnerInstall?(c) OR
       ABS_OwnerSetDisable?(c) OR
       ABS_PhysicalEnable?(c) OR
       ABS_PhysicalDisable?(c) OR
       ABS_PhysicalSetDeactivated?(c) OR
       ABS_SetOperatorAuth?(c) OR
       ABS_TakeOwnership?(c) OR
       ABS_OwnerClear?(c) OR
       ABS_ForceClear?(c) OR
       ABS_DisableOwnerClear?(c) OR
       ABS_PhysicalPresence?(c) OR
       ABS_ResetEstablishmentBit?(c) OR
       ABS_CreateEndorsementKeyPair?(c) OR
       ABS_CreateRevocableEK?(c) OR
       ABS_RevokeTrust?(c)) =>
      permFlags(s) = permFlags(executeCom(s,c));

permFlags_disable_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_OwnerSetDisable?(c) OR
       ABS_PhysicalEnable?(c) OR
       ABS_PhysicalDisable?(c) OR
       ABS_PhysicalSetDeactivated?(c) OR
       ABS_ForceClear?(c) OR
       ABS_OwnerClear?(c) OR
       ABS_RevokeTrust?(c)) =>
      s`permFlags`disable = executeCom(s,c)`permFlags`disable

permFlags_ownership_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
```

```
         ABS_SetOwnerInstall?(c) OR
          ABS_ForceClear?(c) OR
          ABS_OwnerClear?(c) OR
          ABS_RevokeTrust?(c)) =>
       s`permFlags`ownership = executeCom(s,c)`permFlags`ownership

  permFlags_deactivated_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_ForceClear?(c) OR
          ABS_OwnerClear?(c) OR
          ABS_RevokeTrust?(c)) =>
       s`permFlags`deactivated =
          executeCom(s,c)`permFlags`deactivated;

  permFlags_readPubek_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_ForceClear?(c) OR
          ABS_OwnerClear?(c) OR
          ABS_RevokeTrust?(c) OR
          ABS_TakeOwnership?(c)) =>
       s`permFlags`readPubek = executeCom(s,c)`permFlags`readPubek;

  permFlags_disableOC_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_ForceClear?(c) OR
          ABS_OwnerClear?(c) OR
          ABS_RevokeTrust?(c) OR
          ABS_DisableOwnerClear?(c)) =>
       s`permFlags`disableOwnerClear =
executeCom(s,c)`permFlags`disableOwnerClear;

  permFlags_allowMaint_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_ForceClear?(c) OR
          ABS_OwnerClear?(c) OR
          ABS_RevokeTrust?(c)) =>
       s`permFlags`allowMaintenance =
executeCom(s,c)`permFlags`allowMaintenance;

  permFlags_physPresLL_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_PhysicalPresence?(c)) =>
       s`permFlags`physicalPresenceLifetimeLock =
executeCom(s,c)`permFlags`physicalPresenceLifetimeLock;
```

```
  permFlags_physPresHWE_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_PhysicalPresence?(c)) =>
        s`permFlags`physicalPresenceHWEnable =
executeCom(s,c)`permFlags`physicalPresenceHWEnable;

  permFlags_physPresCMDE_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_PhysicalPresence?(c)) =>
        s`permFlags`physicalPresenceCMDEnable =
executeCom(s,c)`permFlags`physicalPresenceCMDEnable;

  permFlags_CEKPUsed_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
    ABS_CreateEndorsementKeyPair?(c) OR
          ABS_CreateRevocableEK?(c)) =>
        s`permFlags`CEKPUsed = executeCom(s,c)`permFlags`CEKPUsed;

  permFlags_FIPS_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c)) =>
        s`permFlags`FIPS = executeCom(s,c)`permFlags`FIPS;

  permFlags_operator_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_SetOperatorAuth?(c) OR
          ABS_ForceClear?(c) OR
          ABS_OwnerClear?(c) OR
          ABS_RevokeTrust?(c)) =>
        s`permFlags`operator = executeCom(s,c)`permFlags`operator;

  permFlags_enableRevokeEK_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_CreateEndorsementKeyPair?(c) OR
          ABS_CreateRevocableEK?(c)) =>
        s`permFlags`enableRevokeEK =
           executeCom(s,c)`permFlags`enableRevokeEK;

  permFlags_nvLocked_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_RevokeTrust?(c)) =>
        s`permFlags`nvLocked = executeCom(s,c)`permFlags`nvLocked;

  permFlags_readSRKPub_unchanged: THEOREM
```

```
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
        ABS_ForceClear?(c) OR
        ABS_OwnerClear?(c) OR
        ABS_RevokeTrust?(c)) =>
      s`permFlags`readSRKPub = executeCom(s,c)`permFlags`readSRKPub;

permFlags_tpmEstablished_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_ResetEstablishmentBit?(c)) =>
      s`permFlags`tpmEstablished =
         executeCom(s,c)`permFlags`tpmEstablished;

permFlags_maintenanceDone_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
        ABS_ForceClear?(c) OR
        ABS_OwnerClear?(c) OR
        ABS_RevokeTrust?(c)) =>
      s`permFlags`maintenanceDone =
         executeCom(s,c)`permFlags`maintenanceDone;

permFlags_disableFullDAL_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
        ABS_ForceClear?(c) OR
        ABS_OwnerClear?(c) OR
        ABS_RevokeTrust?(c)) =>
      s`permFlags`disableFullDALogicInfo =
         executeCom(s,c)`permFlags`disableFullDALogicInfo;

%%%%% PermData
permData_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) or ABS_Init?(c) OR
        ABS_SetOperatorAuth?(c) OR
       ABS_CreateRevocableEK?(c) OR
       ABS_TakeOwnership?(c) OR
       ABS_RevokeTrust?(c) OR
       ABS_ForceClear?(c) OR
       ABS_OwnerClear?(c)) =>
      permData(s) = permData(executeCom(s,c));

permData_tpmProof_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_RevokeTrust?(c) OR
       ABS_ForceClear?(c) OR
       ABS_OwnerClear?(c)) =>
```

```
      s`permData`tpmProof = executeCom(s,c)`permData`tpmProof;

permData_ekReset_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_CreateRevocableEK?(c)) =>
      s`permData`ekReset = executeCom(s,c)`permData`ekReset;

permData_ownerAuth_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
       ABS_TakeOwnership?(c) OR
       ABS_RevokeTrust?(c) OR
       ABS_ForceClear?(c) OR
       ABS_OwnerClear?(c)) =>
      s`permData`ownerAuth = executeCom(s,c)`permData`ownerAuth;

permData_operatorAuth_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c) OR
        ABS_SetOperatorAuth?(c) OR
       ABS_RevokeTrust?(c) OR
       ABS_ForceClear?(c) OR
       ABS_OwnerClear?(c)) =>
      s`permData`operatorAuth =
         executeCom(s,c)`permData`operatorAuth;

permData_pcrAttrib_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c)) =>
      s`permData`pcrAttrib = executeCom(s,c)`permData`pcrAttrib;

%%%%% StanyData
stanyData_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) or ABS_Init?(c) OR
       ABS_OwnerClear?(c) OR ABS_ForceClear?(c) OR
       ABS_RevokeTrust?(c)) =>
      stanyData(s) = stanyData(executeCom(s,c));

%%%%% StanyFlags
stanyFlags_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) or ABS_Init?(c)) =>
      stanyFlags(s) = stanyFlags(executeCom(s,c));

postInit_unchanged: THEOREM
  FORALL (s:tpmAbsState,c:tpmAbsInput) :
    not(ABS_Startup?(c) OR ABS_Init?(c)) =>
      postInitialize(stanyFlags(s)) =
```

```
          postInitialize(stanyFlags(executeCom(s,c)));

  %%%%% StclearData
  stclearData_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) or ABS_Init?(c) OR
          ABS_OwnerClear?(c) OR ABS_ForceClear?(c) OR
          ABS_RevokeTrust?(c)) =>
        stclearData(s) = stclearData(executeCom(s,c));

  %%%%% StclearFlags
  stclearFlags_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) or ABS_Init?(c) OR
          ABS_SetTempDeactivated?(c) OR
          ABS_DisableForceClear?(c) OR
          ABS_PhysicalPresence?(c)) =>
        stclearFlags(s) = stclearFlags(executeCom(s,c));

  disableForceClear_unchanged: THEOREM
    FORALL (s:tpmAbsState,c:tpmAbsInput) :
      not(ABS_Startup?(c) OR ABS_Init?(c) OR
          ABS_DisableForceClear?(c)) =>
        s`stclearFlags`disableForceClear =
          disableForceClear(stclearFlags(executeCom(s,c)));

END invariants
```